LIST OF EXPERIMENTS

1. Basic Operations In R
2. Data Types and Functions In R
3. Data Sorting
4. Finding and Removing Duplicate Records
5. Data Cleaning
6. Data Recording
7. Data Merging
8. Reading and Writing Data In R
9. Data Cleaning and Summarizing with dplyr Package
10. Exploratory Data Analysis
11. Basic Operations on Numpy
12. Computations on Arrays
13. Numpy's Structured Arrays
14. Introducing Pandas Objects
15. Data Indexing and Selection
16. Operating on Data in Pandas
17. Handling Missing Data
18. Hierarchical Indexing
19. Vectorized String Operations
20. Visualization with Matplotlib

LESSON PLAN

Lecture No.	Lecture Topic
	R - Programming
1.	Basic Operations in R
2.	Data Types and Functions in R
3.	Data sorting
4.	Finding and removing Duplicate records
5.	Data Cleaning
6.	Data Recording
7.	Data Merging
8.	Reading and Writing Data in R
9.	Data Cleaning and Summarizing with dplyr package
10.	Exploratory Data Analysis
	Python Programming
1.	Basic operations on Numpy
2.	Computations on Arrays
3.	NumPy's Structured arrays
4.	Introducing Pandas Objects
5.	Data Indexing and Selection
6.	Operating on Data in Pandas
7.	Handling missing data,
8.	Hierarchical Indexing
9.	Vectorized String Operations
10.	Visualization with Matplotlib

Essentials of R Programming

Understand and practice this section thoroughly. This is the building block of your R programming knowledge. If you get this right, you would face less trouble in debugging.

R has five basic or 'atomic' classes of objects. Wait, what is an object ?

Everything you see or create in R is an object. A vector, matrix, data frame, even a variable is an object. R treats it that way. So, R has 5 basic classes of objects. This includes:

1. Character
2. Numeric (Real Numbers)
3. Integer (Whole Numbers)
4. Complex
5. Logical (True / False)

Since these classes are self-explanatory by names, I wouldn't elaborate on that. These classes have attributes. Think of attributes as their 'identifier', a name or number which aptly identifies them. An object can have following attributes:

1. names, dimension names
2. dimensions
3. class
4. length

Attributes of an object can be accessed using *attributes()* function. More on this coming in following section.

Let's understand the concept of object and attributes practically. The most basic object in R is known as vector. You can create an empty vector using *vector()*. Remember, a vector contains object of same class.

For example: Let's create vectors of different classes. We can create vector using *c()* or concatenate command also.

```
> a <- c(1.8, 4.5)   #numeric
> b <- c(1 + 2i, 3 - 6i) #complex
> d <- c(23, 44)   #integer
>  e <- vector("logical", length = 5)
```

Similarly, you can create vector of various classes.

Data Types in R

R has various type of 'data types' which includes vector (numeric, integer etc), matrices, data frames and list. Let's understand them one by one.

Vector: As mentioned above, a vector contains object of same class. But, you can mix objects of different classes too. When objects of different classes are mixed in a list, coercion occurs. This effect causes the objects of different types to 'convert' into one class. For example:

```
>  qt <- c("Time", 24, "October", TRUE, 3.33)  #character
>  ab <- c(TRUE, 24) #numeric
>  cd <- c(2.5, "May") #character
```

To check the class of any object, use *class("vector name")* function.


```
>  class(qt)
 "character"
```

To convert the class of a vector, you can use *as.* command.

```
>  bar <- 0:5
>  class(bar)
>  "integer"
>  as.numeric(bar)
>  class(bar)
>  "numeric"
>  as.character(bar)
>  class(bar)
>  "character"
```

Similarly, you can change the class of any vector. But, you should pay attention here. If you try to convert a "character" vector to "numeric" , NAs will be introduced. Hence, you should be careful to use this command.

List: A list is a special type of vector which contain elements of different data types. For example:

```
> my_list <- list(22, "ab", TRUE, 1 + 2i)
>
my_list
[[1]]
[1] 22
[[2]]
[1] "ab"
[[3]]
[1] TRUE
[[4]]
[1] 1+2i
```

As you can see, the output of a list is different from a vector. This is because, all the objects are of different types. The double bracket [[1]] shows the index of first element and so on. Hence, you can easily extract the element of lists depending on their index. Like this:

```
> my_list[[3]]
> [1] TRUE
```

You can use [] single bracket too. But, that would return the list element with its index number, instead of the result above. Like this:

```
> my_list[3]
> [[1]]
 [1] TRUE
```

Matrices: When a vector is introduced with *row* and *column* i.e. a dimension attribute, it becomes a matrix. A matrix is represented by set of rows and columns. It is a 2 dimensional data structure. It consist of elements of same class. Let's create a matrix of 3 rows and 2 columns:

```
> my_matrix <- matrix(1:6, nrow=3, ncol=2)
>
my_matrix
[,1] [,2]
[1,] 1 4
[2,] 2 5
[3,] 3 6
> dim(my_matrix)
[1] 3 2
> attributes(my_matrix)
$dim
[1] 3 2
```

As you can see, the dimensions of a matrix can be obtained using either *dim()* or *attributes()* command. To extract a particular element from a matrix, simply use the index shown above. For example(try this at your end):

```
> my_matrix[,2]  #extracts second column
> my_matrix[,1]  #extracts first column
> my_matrix[2,]  #extracts second row
> my_matrix[1,]  #extracts first row
```


As an interesting fact, you can also create a matrix from a vector. All you need to do is, assign dimension *dim()*later. Like this:

```
> age <- c(23, 44, 15, 12, 31, 16)
> age
[1] 23 44 15 12 31 16

> dim(age) <- c(2,3)
> age
[,1] [,2] [,3]
[1,] 23 15 31
[2,] 44 12 16
>
class(age)
[1] "matrix"
```

You can also join two vectors using *cbind()* and *rbind()* functions. But, make sure that both vectors have same number of elements. If not, it will return NA values.

```
> x <- c(1, 2, 3, 4, 5, 6)
> y <- c(20, 30, 40, 50, 60)
> cbind(x, y)
> cbind(x,
y) xy
[1,] 1 20
[2,] 2 30
[3,] 3 40
[4,] 4 50
[5,] 5 60
[6,] 6 70
> class(cbind(x, y))
[1] "matrix"
```

Data Frame: This is the most commonly used member of data types family. It is used to store tabular data. It is different from matrix. In a matrix, every element must have same class. But, in a data frame, you can put list of vectors containing different classes. This means, every column of a data frame acts like a list. Every time you will read data in R, it will be stored in the form of a data frame. Hence, it is important to understand the majorly used commands on data frame:

```
> df <- data.frame(name = c("ash","jane","paul","mark"), score = c(67,56,87,91))
> df
name score
1 ash 67
2 jane  56
3 paul  87
4 mark 91


>
```



```
dim(df)
[1] 4 2
```



```
> str(df)
'data.frame': 4 obs. of 2 variables:
$ name : Factor w/ 4 levels "ash","jane","mark",..: 1 2 4 3
$ score: num 67 56 87 91

>
nrow(df)
[1] 4

>
ncol(df)
[1] 2
```

Let's understand the code above. *df* is the name of data frame. *dim()* returns the dimension of data frame as 4 rows and 2 columns. *str()* returns the structure of a data frame i.e. the list of variables stored in the data frame. *nrow()* and *ncol()* return the number of rows and number of columns in a data set respectively.

Here you see "name" is a factor variable and "score" is numeric. In data science, a variable can be categorized into two types: Continuous and Categorical.

Continuous variables are those which can take any form such as 1, 2, 3.5, 4.66 etc. **Categorical variables** are those which takes only discrete values such as 2, 5, 11, 15 etc. In R, categorical values are represented by factors. In df, name is a factor variable having 4 unique levels. Factor or categorical variable are specially treated in a data set. For more explanation, click here. Similarly, you can find techniques to deal with continuous variables here.

Let's now understand the concept of **missing values** in R. This is one of the most painful yet crucial part of predictive modeling. You must be aware of all techniques to deal with them. The complete explanation on such techniques is provided here.

Missing values in R are represented by *NA* and *NaN*. Now we'll check if a data set has missing values (using the same data frame *df*).

```
> df[1:2,2] <- NA #injecting NA at 1st, 2nd row and 2nd column of df
> df
name score
1  ash  NA
2  jane  NA
3 paul 87
4 mark 91

> is.na(df) #checks the entire data set for NAs and return logical
output name score
[1,] FALSE TRUE
[2,] FALSE TRUE
[3,] FALSE FALSE
[4,] FALSE FALSE

> table(is.na(df)) #returns a table of logical
output FALSE TRUE
```

6 2

> df[!complete.cases(df),] #returns the list of rows having missing values
name score
1 ash NA
2 jane NA

Missing values hinder normal calculations in a data set. For example, let's say, we want to compute the mean of score. Since there are two missing values, it can't be done directly. Let's see:

mean(df$score)
[1] NA
> mean(df$score, na.rm =
TRUE) [1] 89

The use of *na.rm = TRUE* parameter tells R to ignore the NAs and compute the mean of remaining values in the selected column (score). To remove rows with NA values in a data frame, you can use *na.omit*:

> new_df <- na.omit(df)
> new_df
name score
3 paul 87
4 mark 91

Useful R Packages

Out of ~7800 packages listed on CRAN, I've listed some of the most powerful and commonly used packages in predictive modeling in this article. Since, I've already explained the method of installing packages, you can go ahead and install them now. Sooner or later you'll need them.

Importing Data: R offers wide range of packages for importing data available in any format such as .txt, .csv, .json, .sql etc. To import large files of data quickly, it is advisable to install and use *data.table*, *readr*, *RMySQL*, *sqldf*, *jsonlite*.

Data Visualization: R has in built plotting commands as well. They are good to create simple graphs. But, becomes complex when it comes to creating advanced graphics. Hence, you should install *ggplot2*.

Data Manipulation: R has a fantastic collection of packages for data manipulation. These packages allows you to do basic & advanced computations quickly. These packages are *dplyr*, *plyr*, *tidyr*, *lubridate*, *stringr*. Check out this complete tutorial on data manipulation packages in R.

Modeling / Machine Learning: For modeling, *caret* package in R is powerful enough to cater to every need for creating machine learning model. However, you can install packages algorithms wise such as *randomForest*, *rpart*, *gbm* etc

Note: *I've only mentioned the commonly used packages. You might like to check this interesting infographic on complete list of useful R packages.*

Exploratory Data Analysis in R

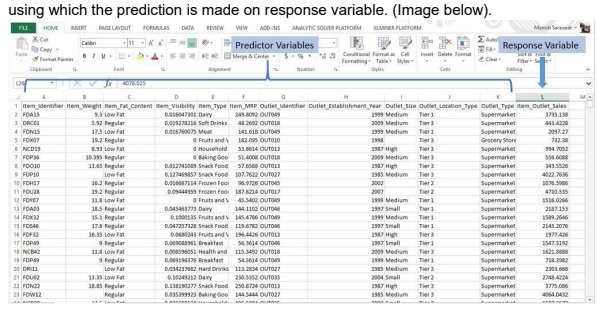
From this section onwards, we'll dive deep into various stages of predictive modeling. Hence, make sure you understand every aspect of this section. In case you find anything difficult to understand, ask me in the comments section below.

Data Exploration is a crucial stage of predictive model. You can't build great and practical models unless you learn to explore the data from begin to end. This stage forms a concrete foundation for data manipulation (the very next stage). Let's understand it in R.

In this tutorial, I've taken the data set from Big Mart Sales Prediction. Before we start, you must get familiar with these terms:

Response Variable (a.k.a Dependent Variable): In a data set, the response variable (y) is one on which we make predictions. In this case, we'll predict 'Item_Outlet_Sales'. (Refer to image shown below)

Predictor Variable (a.k.a Independent Variable): In a data set, predictor variables (Xi) are those using which the prediction is made on response variable. (Image below).



Train Data: The predictive model is always built on train data set. An intuitive way to identify the train data is, that it always has the 'response variable' included.

Test Data: Once the model is built, it's accuracy is 'tested' on test data. This data always contains less number of observations than train data set. Also, it does not include 'response variable'.

Right now, you should download the data set. Take a good look at train and test data. Cross check the information shared above and then proceed.

Let's now begin with **importing and exploring data**.

#working directory
path <- ".../Data/BigMartSales"

#set working directory
setwd(path)

As a beginner, I'll advise you to keep the train and test files in your working directly to avoid unnecessary directory troubles. Once the directory is set, we can easily import the .csv files using commands below.

#Load Datasets
train <- read.csv("Train_UWu5bXk.csv")
test <- read.csv("Test_u94Q5KV.csv")

In fact, even prior to loading data in R, it's a good practice to look at the data in Excel. This helps in strategizing the complete prediction modeling process. To check if the data set has been loaded successfully, look at R environment. The data can be seen there. Let's explore the data quickly.

#check dimesions (number of row & columns) in data set
>
dim(train)
[1] 8523 12

> dim(test)
[1] 5681 11

We have 8523 rows and 12 columns in train data set and 5681 rows and 11 columns in data set. This makes sense. Test data should always have one column less (mentioned above right?). Let's get deeper in train data set now.

#check the variables and their types in train
> str(train)
'data.frame': 8523 obs. of 12 variables:
$ Item_Identifier : Factor w/ 1559 levels "DRA12","DRA24",..: 157 9 663 1122 1298 759 697 739 441 991 ...
$ Item_Weight : num 9.3 5.92 17.5 19.2 8.93 ...
$ Item_Fat_Content : Factor w/ 5 levels "LF","low fat",..: 3 5 3 5 3 5 5 3 5 5 ...
$ Item_Visibility : num 0.016 0.0193 0.0168 0 0 ...
$ Item_Type : Factor w/ 16 levels "Baking Goods",..: 5 15 11 7 10 1 14 14 6 6 ...
$ Item_MRP : num 249.8 48.3 141.6 182.1 53.9 ...
$ Outlet_Identifier : Factor w/ 10 levels "OUT010","OUT013",..: 10 4 10 1 2 4 2 6 8 3 ...
$ Outlet_Establishment_Year: int 1999 2009 1999 1998 1987 2009 1987 1985 2002 2007 ...
$ Outlet_Size : Factor w/ 4 levels "","High","Medium",..: 3 3 3 1 2 3 2 3 1 1 ...
$ Outlet_Location_Type : Factor w/ 3 levels "Tier 1","Tier 2",..: 1 3 1 3 3 3 3 3 2 2 ...
$ Outlet_Type : Factor w/ 4 levels "Grocery Store",..: 2 3 2 1 2 3 2 4 2 2 ...
$ Item_Outlet_Sales : num 3735 443 2097 732 995 ...

Let's do some quick data exploration.

To begin with, I'll first check if this data has missing values. This can be done by using:

>
table(is.na(train))
FALSE TRUE
100813 1463

In train data set, we have 1463 missing values. Let's check the variables in which these values are missing. It's important to find and locate these missing values. Many data scientists have repeatedly advised beginners to pay close attention to missing value in data exploration stages.

> colSums(is.na(train))
Item_Identifier Item_Weight
0 1463
Item_Fat_Content Item_Visibility

0 0
Item_Type Item_MRP

0 0
Outlet_Identifier Outlet_Establishment_Year
0 0
Outlet_Size Outlet_Location_Type
0 0
Outlet_Type Item_Outlet_Sales
0 0

Hence, we see that column Item_Weight has 1463 missing values. Let's get more inferences from this data.

```
> summary(train)
```

Here are some quick inferences drawn from variables in train data set:

1. Item_Fat_Content has mis-matched factor levels.
2. Minimum value of item_visibility is 0. Practically, this is not possible. If an item occupies shelf space in a grocery store, it ought to have some visibility. We'll treat all 0's as missing values.
3. Item_Weight has 1463 missing values (already explained above).
4. Outlet_Size has a unmatched factor levels.

These inference will help us in treating these variable more accurately.

Data Manipulation with dplyr

The dplyr package is one of the most powerful and popular package in R.

This package was written by the most popular R programmer Hadley Wickham who has written many useful R packages such as ggplot2, tidyr etc. This post includes several examples and tips of how to use dplyr package for cleaning and transforming data. It's a complete tutorial on data manipulation and data wrangling with R.

What is dplyr?

The dplyr is a powerful R-package to manipulate, clean and summarize unstructured data. In short, it makes data exploration and data manipulation easy and fast in R.

What's special about dplyr?

The package "dplyr" comprises many functions that perform mostly used data manipulation operations such as applying filter, selecting specific columns, sorting data, adding or deleting columns and aggregating data.

Another most important advantage of this package is that it's very easy to learn and use dplyr functions. Also easy to recall these functions. For example, **filter()** is used to filter rows.

dplyr vs. Base R Functions

dplyr functions process faster than base R functions. It is because dplyr functions were written in a computationally efficient manner. They are also more stable in the syntax and better supports data frames than vectors.

SQL Queries vs. dplyr

People have been utilizing SQL for analyzing data for decades. Every modern data analysis software such as Python, R, SAS etc supports SQL commands. But SQL was never designed to perform data analysis. It was rather designed for querying and managing data. There are many data analysis operations where SQL fails or makes simple things difficult. For example, calculating median for multiple variables, converting wide format data to long format etc. Whereas, dplyr package was designed to do data
analysis.

The names of dplyr functions are similar to SQL commands such as **select()** for selecting variables, **group_by()** - group data by grouping variable, join() - joining two data sets. Also includes **inner_join()** and **left_join()**. It also supports sub queries for which SQL was popular for.

How to install and load dplyr package

To install the dplyr package, type the following command.

```
install.packages("dplyr")
```

To load dplyr package, type the command below

```
library(dplyr)
```

Important dplyr Functions to remember

dplyr Function Description Equivalent SQL

select() Selecting columns (variables) SELECT
filter() Filter (subset) rows. WHERE
group_by() Group the data GROUP BY
summarise() Summarise (or aggregate) data
arrange() Sort the data ORDER BY
join() Joining data frames (tables) JOIN
mutate() Creating New Variables COLUMN
ALIAS

Data : Income Data by States

In this tutorial, we are using the following data which contains income generated by states from year 2002 to 2015. **Note :** This data do not contain actual income figures of the states.

This dataset contains 51 observations (rows) and 16 variables (columns).

The snapshot of first 6 rows of the dataset is shown below.

Index State Y2002 Y2003 Y2004 Y2005 Y2006 Y2007 Y2008 Y2009
1 A Alabama 1296530 1317711 1118631 1492583 1107408 1440134 1945229 1944173
2 A Alaska 1170302 1960378 1818085 1447852 1861639 1465841 1551826 1436541
3 A Arizona 1742027 1968140 1377583 1782199 1102568 1109382 1752886 1554330
4 A Arkansas 1485531 1994927 1119299 1947979 1669191 1801213 1188104 1628980
5 C California 1685349 1675807 1889570 1480280 1735069 1812546 1487315 1663809
6 C Colorado 1343824 1878473 1886149 1236697 1871471 1814218 1875146 1752387

Y2010 Y2011 Y2012 Y2013 Y2014 Y2015
1 1237582 1440756 1186741 1852841 1558906 1916661
2 1629616 1230866 1512804 1985302 1580394 1979143
3 1300521 1130709 1907284 1363279 1525866 1647724
4 1669295 1928238 1216675 1591896 1360959 1329341
5 1624509 1639670 1921845 1156536 1388461 1644607
6 1913275 1665877 1491604 1178355 1383978 1330736

Download the Dataset

How to load Data

Submit the following code. ***Change the file path in the code below.***

```
mydata = read.csv("C:\\Users\\Deepanshu\\Documents\\sampledata.csv")
```

Example 1 : Selecting Random N Rows

The **sample_n** function selects random rows from a data frame (or table).
The second parameter of the function tells R the number of rows to select.

```
sample_n(mydata,3)
```

Index State Y2002 Y2003 Y2004 Y2005 Y2006 Y2007 Y2008 Y2009
2 A Alaska 1170302 1960378 1818085 1447852 1861639 1465841 1551826 1436541
8 D Delaware 1330403 1268673 1706751 1403759 1441351 1300836 1762096 1553585
33 N New York 1395149 1611371 1170675 1446810 1426941 1463171 1732098 1426216
Y2010 Y2011 Y2012 Y2013 Y2014 Y2015
2 1629616 1230866 1512804 1985302 1580394 1979143
8 1370984 1318669 1984027 1671279 1803169 1627508
33 1604531 1683687 1500089 1718837 1619033 1367705

Example 2 : Selecting Random Fraction of Rows

The **sample_frac** function returns randomly N% of rows. In the example below, it returns randomly 10% of rows.

```
sample_frac(mydata,0.1)
```

Example 3 : Remove Duplicate Rows based on all the variables (Complete Row)

The **distinct function** is used to eliminate duplicates.

```
x1 = distinct(mydata)
```

In this dataset, there is not a single duplicate row so it returned same number of rows as in mydata

Example 4 : Remove Duplicate Rows based on a variable

The **.keep_all** function is used to retain all other variables in the output data frame.

```
x2 = distinct(mydata, Index, .keep_all= TRUE)
```

Example 5 : Remove Duplicates Rows based on multiple variables

In the example below, we are using two variables - **Index, Y2010** to determine uniqueness.

```
x2 = distinct(mydata, Index, Y2010, .keep_all= TRUE)
```

select( ) Function

4/24

It is used to select only desired variables.

select() syntax : select(data , ...)
data : Data Frame
.... : Variables by name or by function

Example 6 : Selecting Variables (or Columns)

Suppose you are asked to select only a few variables. The code below selects variables "Index", columns from "State" to "Y2008".

```
mydata2 = select(mydata, Index, State:Y2008)
```

Example 7 : Dropping Variables

The **minus sign** before a variable tells R to drop the variable.

```
mydata = select(mydata, -Index, -State)
```

The above code can also be written like :

```
mydata = select(mydata, -c(Index,State))
```

Example 8 : Selecting or Dropping Variables starts with 'Y'

The **starts_with()** function is used to select variables starts with an alphabet.

```
mydata3 = select(mydata, starts_with("Y"))
```

Adding a negative sign before starts_with() implies dropping the variables starts with 'Y'

```
mydata33 = select(mydata, -starts_with("Y"))
```


The following functions helps you to select variables based on their names.

Helpers Description

starts_with() Starts with a prefix
ends_with() Ends with a prefix
contains() Contains a literal string
matches() Matches a regular expression

Output

num_range() Numerical range like x01, x02, x03.
one_of() Variables in character vector.
everything() All variables.

Example 9 : Selecting Variables contain 'I' in their names

```
mydata4 = select(mydata, contains("I"))
```

Example 10 : Reorder Variables

The code below keeps variable **'State'** in the front and the remaining variables follow that.

```
mydata5 = select(mydata, State, everything())
```

New order of variables are displayed below -

[1] "State" "Index" "Y2002" "Y2003" "Y2004" "Y2005" "Y2006" "Y2007" "Y2008" "Y2009"
[11] "Y2010" "Y2011" "Y2012" "Y2013" "Y2014" "Y2015"

rename( ) Function

It is used to change variable name.

rename() syntax : rename(data , new_name = old_name)
data : Data Frame
new_name : New variable name you want to keep
old_name : Existing Variable Name

Example 11 : Rename Variables

The rename function can be used to rename variables.
In the following code, we are renaming **'Index'** variable to **'Index1'**.

```
mydata6 = rename(mydata, Index1=Index)
```

filter( ) Function

It is used to subset data with matching logical conditions.

6/24

filter() syntax : filter(data , ...)
data : Data Frame
.... : Logical Condition

Example 12 : Filter Rows

Suppose you need to subset data. You want to filter rows and retain only those values in which Index is equal to A.

```
mydata7 = filter(mydata, Index == "A")
```

Index State Y2002 Y2003 Y2004 Y2005 Y2006 Y2007 Y2008 Y2009
1 A Alabama 1296530 1317711 1118631 1492583 1107408 1440134 1945229 1944173
2 A Alaska 1170302 1960378 1818085 1447852 1861639 1465841 1551826 1436541
3 A Arizona 1742027 1968140 1377583 1782199 1102568 1109382 1752886 1554330
4 A Arkansas 1485531 1994927 1119299 1947979 1669191 1801213 1188104 1628980
Y2010 Y2011 Y2012 Y2013 Y2014 Y2015
1 1237582 1440756 1186741 1852841 1558906 1916661
2 1629616 1230866 1512804 1985302 1580394 1979143
3 1300521 1130709 1907284 1363279 1525866 1647724
4 1669295 1928238 1216675 1591896 1360959 1329341

Example 13 : Multiple Selection Criteria

The **%in%** operator can be used to select multiple items. In the following program, we are telling R to select rows against 'A' and 'C' in column 'Index'.

```
mydata7 = filter(mydata6, Index %in% c("A", "C"))
```

Example 14 : 'AND' Condition in Selection Criteria

Suppose you need to apply 'AND' condition. In this case, we are picking data for 'A' and 'C' in the column 'Index' and income greater than 1.3 million in Year 2002.

```
mydata8 = filter(mydata6, Index %in% c("A", "C") & Y2002 >= 1300000 )
```

Example 15 : 'OR' Condition in Selection Criteria

7/24

Output

The 'I' denotes OR in the logical condition. It means any of the two conditions.

```
mydata9 = filter(mydata6, Index %in% c("A", "C") | Y2002 >= 1300000)
```

Example 16 : NOT Condition

The **"!"** sign is used to reverse the logical condition.

```
mydata10 = filter(mydata6, !Index %in% c("A", "C"))
```

Example 17 : CONTAINS Condition

The **grepl function** is used to search for pattern matching. In the following code, we are looking for records wherein column **state** contains **'Ar'** in their name.

```
mydata10 = filter(mydata6, grepl("Ar", State))
```

summarise( ) Function

It is used to summarize data.

summarise() syntax : summarise(data ,.....)

data : Data Frame

..... : Summary Functions such as mean, median etc

Example 18 : Summarize selected variables

In the example below, we are calculating mean and median for the variable Y2015.

```
summarise(mydata, Y2015_mean = mean(Y2015),
Y2015_med=median(Y2015))
```

Example 19 : Summarize Multiple Variables

In the following example, we are calculating number of records, mean and median for variables Y2005 and Y2006. The **summarise_at** function allows us to

8/24

select multiple variables by their names.

```
summarise_at(mydata, vars(Y2005, Y2006), funs(n(), mean, median))
```

Output

Example 20 : Summarize with Custom Functions

We can also use custom functions in the summarise function. In this case, we are computing the number of records, number of missing values, mean and median for variables Y2011 and Y2012. The **dot (.)** denotes each variables specified in the second argument of the function.

```
summarise_at(mydata, vars(Y2011, Y2012),
funs(n(), missing = sum(is.na(.)), mean(., na.rm = TRUE), median(.,na.rm
= TRUE)))
```

Summarize : Output

How to apply Non-Standard Functions

Suppose you want to subtract mean from its original value and then calculate variance of it.

```
set.seed(222)
mydata <- data.frame(X1=sample(1:100,100), X2=runif(100))
summarise_at(mydata,vars(X1,X2), function(x) var(x - mean(x)))
```


X1 X2

1 841.6667 0.08142161

Example 21 : Summarize all Numeric Variables

The **summarise_if** function allows you to summarise conditionally.

```
summarise_if(mydata, is.numeric, funs(n(),mean,median))
```

Alternative Method :

First, store data for all the numeric variables

```
numdata = mydata[sapply(mydata,is.numeric)]
```

Second, the **summarise_all** function calculates summary statistics for all the columns in a data frame

```
summarise_all(numdata, funs(n(),mean,median))
```

Example 22 : Summarize Factor Variable

We are checking the **number of levels/categories** and **count of missing observations** in a categorical (factor) variable.

```
summarise_all(mydata["Index"], funs(nlevels(.), nmiss=sum(is.na(.))))
```

nlevels nmiss

1 19 0

arrange() function :

Use : Sort data

Syntax

```
arrange(data_frame, variable(s)_to_sort)
```

or

```
data_frame %>% arrange(variable(s)_to_sort)
```

To sort a variable in descending order, use **desc(x)**.

Example 23 : Sort Data by Multiple Variables

The default sorting order of **arrange() function** is ascending. In this example, we are sorting data by multiple variables.

```
arrange(mydata, Index, Y2011)
```

Suppose you need to sort one variable by descending order and other variable by ascending oder


```
arrange(mydata, desc(Index), Y2011)
```

10/24

Pipe Operator %>%

It is important to understand the pipe (%>%) operator before knowing the other functions of dplyr package. dplyr utilizes pipe operator from another package **(magrittr)**.

It allows you to write sub-queries like we do it in sql.

Note : All the functions in dplyr package can be used **without** the pipe operator. The question arises **"Why to use pipe operator %>%". The answer is** it lets to wrap multiple functions together with the use of %>%.

Syntax :

```
filter(data_frame, variable == value)
```

or

```
data_frame %>% filter(variable == value)
```

The %>% is NOT restricted to filter function. It can be used with any function.

Example :

The code below demonstrates the usage of pipe %>% operator. In this example, we are selecting 10 random observations of two variables "Index" "State" from the data frame "mydata".

```
dt = sample_n(select(mydata, Index, State),10)
```

or

```
dt = mydata %>% select(Index, State) %>% sample_n(10)
```

group_by() function :

Use : Group data by categorical variable

Syntax :

```
group_by(data, variables)
```

or

```
data %>% group_by(variables)
```

Example 24 : Summarise Data by Categorical Variable

We are calculating count and mean of variables Y2011 and Y2012 by variable Index.

```
t = summarise_at(group_by(mydata, Index), vars(Y2011, Y2012), funs(n(),
mean(., na.rm = TRUE)))
```

The above code can also be written like

```
t = mydata %>% group_by(Index) %>%
summarise_at(vars(Y2011:Y2015), funs(n(), mean(., na.rm = TRUE)))
```

Index Y2011_n Y2012_n Y2013_n Y2014_n Y2015_n Y2011_mean Y2012_mean
A 4 4 4 4 4 1432642 1455876
C 3 3 3 3 3 1750357 1547326
D 2 2 2 2 2 1336059 1981868
F 1 1 1 1 1 1497051 1131928
G 1 1 1 1 1 1851245 1850111
H 1 1 1 1 1 1902816 1695126
I 4 4 4 4 4 1690171 1687056
K 2 2 2 2 2 1489353 1899773
L 1 1 1 1 1 1210385 1234234
M 8 8 8 8 8 1582714 1586091
N 8 8 8 8 8 1448351 1470316
O 3 3 3 3 3 1882111 1602463
P 1 1 1 1 1 1483292 1290329
R 1 1 1 1 1 1781016 1909119
S 2 2 2 2 2 1381724 1671744
T 2 2 2 2 2 1724080 1865787
U 1 1 1 1 1 1288285 1108281
V 2 2 2 2 2 1482143 1488651
W 4 4 4 4 4 1711341 1660192

do() function :

Use : Compute within groups

Syntax :


```
do(data_frame, expressions_to_apply_to_each_group)
```

Note : *The **dot (.)** is required to refer to a data frame.*

12/24

Output

Example 25 : Filter Data within a Categorical Variable

Suppose you need to pull top 2 rows from 'A', 'C' and 'I' categories of variable Index.

```
t = mydata %>% filter(Index %in% c("A", "C","I")) %>% group_by(Index)
%>%
do(head( . , 2))
```

Example 26 : Selecting 3rd Maximum Value by Categorical Variable

We are calculating third maximum value of variable Y2015 by variable Index. The following code first selects only two variables Index and Y2015. Then it filters the variable Index with 'A', 'C' and 'I' and then it groups the same variable and sorts the variable Y2015 in descending order. At last, it selects the third row.

```
t = mydata %>% select(Index, Y2015) %>%
filter(Index %in% c("A", "C","I")) %>%
group_by(Index) %>%
do(arrange(.,desc(Y2015))) %>% slice(3)
```

The **slice() function** is used to select rows by position.

Using Window Functions

Like SQL, dplyr uses window functions that are used to subset data within a group. It returns a vector of values. We could use **min_rank() function** that calculates rank in the preceding example,

```
t = mydata %>% select(Index, Y2015) %>%
filter(Index %in% c("A", "C","I")) %>%
group_by(Index) %>%
filter(min_rank(desc(Y2015)) == 3)
```

Index Y2015
1 A 1647724
2 C 1330736
3 I 1583516

Example 27 : Summarize, Group and Sort Together

In this case, we are computing mean of variables Y2014 and Y2015 by variable Index. Then sort the result by calculated mean variable Y2015.

```
t = mydata %>%
group_by(Index)%>%
summarise(Mean_2014 = mean(Y2014, na.rm=TRUE),
Mean_2015 = mean(Y2015, na.rm=TRUE)) %>%
arrange(desc(Mean_2015))
```

mutate() function :

Use : Creates new variables

Syntax :

```
mutate(data_frame, expression(s) )
```

or

```
data_frame %>% mutate(expression(s))
```

Example 28 : Create a new variable

The following code calculates division of Y2015 by Y2014 and name it "change".

```
mydata1 = mutate(mydata, change=Y2015/Y2014)
```

Example 29 : Multiply all the variables by 1000

It creates new variables and name them with suffix "_new".

14/24

Output

Output

```
mydata11 = mutate_all(mydata, funs("new" = .* 1000))
```

The output shown in the image above is truncated due to high number of variables.

Note - The above code returns the following error messages -

Warning messages:

1: In Ops.factor(c(1L, 1L, 1L, 1L, 2L, 2L, 2L, 3L, 3L, 4L, 5L, 6L, :
'*' not meaningful for factors
2: In Ops.factor(1:51, 1000) : '*' not meaningful for factors

It implies you are multiplying 1000 to string(character) values which are stored as factor variables. These variables are 'Index', 'State'. It does not make sense to apply multiplication operation on character variables. For these two variables, it creates newly created variables which contain only NA.

Solution : See **Example 45** -Apply multiplication on only numeric variables

Example 30 : Calculate Rank for Variables

Suppose you need to calculate rank for variables Y2008 to Y2010.

```
mydata12 = mutate_at(mydata, vars(Y2008:Y2010), funs(Rank=min_rank(.)))
```

By default, **min_rank()** assigns 1 to the smallest value and high number to the largest value. In case, you need to assign rank 1 to the largest value of a variable, use **min_rank(desc(.))**

```
mydata13 = mutate_at(mydata,
vars(Y2008:Y2010),
funs(Rank=min_rank(desc(.))))
```

Example 31 : Select State that generated highest income among the variable 'Index'

```
out = mydata %>% group_by(Index) %>% filter(min_rank(desc(Y2015)) ==
1) %>%
select(Index, State, Y2015)
```

Index State Y2015
1 A Alaska 1979143
2 C Connecticut 1718072
3 D Delaware 1627508
4 F Florida 1170389
5 G Georgia 1725470
6 H Hawaii 1150882
7 I Idaho 1757171
8 K Kentucky 1913350
9 L Louisiana 1403857
10 M Missouri 1996005

11 N New Hampshire 1963313
12 O Oregon 1893515
13 P Pennsylvania 1668232
14 R Rhode Island 1611730
15 S South Dakota 1136443
16 T Texas 1705322
17 U Utah 1729273
18 V Virginia 1850394
19 W Wyoming 1853858

Python Numpy Array Tutorial

NumPy is, just like SciPy, Scikit-Learn, Pandas, etc. one of the packages that you just can't miss when you're learning data science, mainly because this library provides you with an array data structure that holds some benefits over Python lists, such as: being more compact, faster access in reading and writing items, being more convenient and more efficient.

NumPy is a Python library that is the core library for scientific computing in Python. It contains a collection of tools and techniques that can be used to solve on a computer mathematical models of problems in Science and Engineering. One of these tools is a high-performance multidimensional array object that is a powerful data structure for efficient computation of arrays and matrices. To work with these arrays, there's a huge amount of high-level mathematical functions operate on these matrices and arrays.

Creation of Arrays

What is an array

A grid that contains values of the same type

Creation of arrays

1-d array

In [6]:

```
import numpy as np
```

In [2]:

```
a = np.array([12,23,34,56,67,78,89])
```

2-d array

In [18]:

```
b = np.array([[1,2],[3,4]])
```

multi-dimensional array

In [19]:

```
c = np.array([[1,2,3],[4,5,6],[7,8,9]])
```


Print an array

In [5]:

```
print(a)
```

```
[12 23 34 56 67 78 89]
```

In [6]:

```
a
```

Out[6]:

```
array([12, 23, 34, 56, 67, 78, 89])
```

In [5]:

```
print(b)
```

```
[[1 2]
 [3 4]]
```

In [7]:

```
print(c)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Creation of various types of arrays

For some, such as **np.ones()**, **np.random.random(), np.empty(), np.full()** or **np.zeros()** the only thing that you need to do in order to make arrays with ones or zeros is pass the shape of the array that you want to make. As an option to **np.ones()** and **np.zeros()**, you can also specify the data type. In case of **np.full()**, you also have to specify the constant value that you want to insert into the array.

In [9]:

```
np.ones((3,4))
```

Out[9]:

```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

In [10]:

```
np.zeros((3,4))
```

Out[10]:

```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

In [11]:


```
np.empty((3,4))
```

Out[11]:

```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

In [14]:

```
np.random.random((3,4))
```

Out[14]:

```
array([[0.35828038, 0.76453824, 0.83428902, 0.16816453],
       [0.21407214, 0.10989867, 0.28033833, 0.42555466],
       [0.71751008, 0.17931965, 0.6932923 , 0.05775626]])
```

In [16]:

```
np.ones((3,4), dtype = np.int32)
```

Out[16]:

```
array([[1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1]])
```

In [29]:

```
# Create a full array
np.full((2,2),7)
```

Out[29]:

```
array([[7, 7],
       [7, 7]])
```

In [30]:

```
# Create a full array
np.full((4,2),24)
```

Out[30]:

```
array([[24, 24],
       [24, 24],
       [24, 24],
       [24, 24]])
```

In [37]:

```
# Create a diagonal matrix or array
np.eye(5)
```


Out[37]:

```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

In [38]:

```
# Create a identity matrix or array
np.identity(3)
```

Out[38]:

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

In [23]:

```
# Create a diagonal matrix or array having diagonal elements as 1,2,3 and 4
np.diag(np.array([1, 2, 3, 4]))
```

Out[23]:

```
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

In [24]:

```
# Create a diagonal matrix or array having diagonal elements as 51,22,36,10 and 423
np.diag(np.array([51,22,36,10,423]))
```

Out[24]:

```
array([[ 51,   0,   0,   0,   0],
       [  0,  22,   0,   0,   0],
       [  0,   0,  36,   0,   0],
       [  0,   0,   0,  10,   0],
       [  0,   0,   0,   0, 423]])
```

Numpy array attributes

- The **data** pointer indicates the memory address of the first byte in the array,
- The data type or **dtype** pointer describes the kind of elements that are contained within the array,
- The **shape** indicates the shape of the array, and

- The **strides** are the number of bytes that should be skipped in memory to go to the next element. If your strides are (10,1), you need to proceed one byte to get to the next column and 10 bytes to locate the next row.

Examples

In [7]:
```
a = np.array([12,23,34,56,67,78,89])
print(a)
```
[12 23 34 56 67 78 89]

In [41]:
```
print(a.data)
```
<memory at 0x000001FBDBFBBA08>

In [42]:
```
print(a.dtype)
```
int32

In [43]:
```
print(a.shape)
```
(7,)

In [44]:
```
print(a.strides)
```
(4,)

Some more attributes

In [45]:
```
# Print the number of dimensions of a
print(a.ndim)
```
1

In [9]:
```
# Print the number of elements of a
print(a.size)
```
7

In [10]:
```
# Print the length of one array element in bytes
print(a.itemsize)
```
4

In [21]:
```
# print the length of array a
```



```
print(len(a))
```
7

In [22]:
```
# print the length of array b
print(b)
print(len(b))      # returns the size of the first dimension
```
[[1 2]
 [3 4]]
2

In [159]:
```
# Change the data type of array b from int to float
print(b)
print(b.dtype)
b = b.astype(float)
print(b)
print(b.dtype)
```
[4 5 6]
int32
[4. 5. 6.]
float64

Some functions of Numpy package

numpy.arange()

About :

arange([start,] stop[, step,][, dtype]) : Returns an array with evenly spaced elements as per the interval. The interval mentioned is half opened i.e. [Start, Stop)

Parameters :

start : [optional] start of interval range. By default start = 0
stop : end of interval range
step : [optional] step size of interval. By default step size = 1, For any output out, this is the distance between two adjacent values, out[i+1] - out[i].
dtype : type of output array

Return:

Array of evenly spaced values.
Length of array being generated = Ceil((Stop - Start) / Step)

In [61]:
```
# Create an array of elements from 0 to 10
np.arange(11)
```
Out[61]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [62]:
```
# Create an array of elements from 0 to 25
np.arange(26)
```
Out[62]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25])
```

In [64]:
```
# Create an array of elements from 10 to 20
np.arange(10,21)  # 10 is included but 21 is excluded
```
Out[64]:
```
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20])
```

In [65]:
```
# Create an array of elements from 15 to 25
np.arange(15,26)  # 10 is included but 21 is excluded
```
Out[65]:
```
array([15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25])
```

In [21]:
```
# Create an array of evenly-spaced values with difference 5
np.arange(10,25,5)
```
Out[21]:
```
array([10, 15, 20])
```

In [22]:
```
# Create an array of evenly-spaced values with difference 3
np.arange(1,25,3)
```
Out[22]:
```
array([ 1,  4,  7, 10, 13, 16, 19, 22])
```

In [68]:
```
# Create an array of evenly-spaced values with difference 3 - specify dtype as float
```



```
# first method
np.arange(1,25,3.0)
```
Out[68]:
```
array([ 1.,  4.,  7., 10., 13., 16., 19., 22.])
```

In [76]:
```
# Create an array of evenly-spaced values with difference 3 - specify dtype as float
# second method
np.arange(1,25,3, dtype = np.float32)
```
Out[76]:
```
array([ 1.,  4.,  7., 10., 13., 16., 19., 22.], dtype=float32)
```

In [77]:
```
# to remove dtype = float32 from above output
print(np.arange(1,25,3, dtype = np.float32))
```
[1. 4. 7. 10. 13. 16. 19. 22.]

In [27]:
```
# Create an array of evenly-spaced values
np.linspace(1,9,9)
```
Out[27]:
```
array([1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

In [28]:
```
# Create an array of evenly-spaced values
np.linspace(1,9,18)
```
Out[28]:
```
array([1.        , 1.47058824, 1.94117647, 2.41176471, 2.88235294,
       3.35294118, 3.82352941, 4.29411765, 4.76470588, 5.23529412,
       5.70588235, 6.17647059, 6.64705882, 7.11764706, 7.58823529,
       8.05882353, 8.52941176, 9.        ])
```

Note: Practice above attributes on multi dimensional arrays

Array Indexing: Accessing Single Elements

One Dimensional Indexing

Generally, indexing works just like you would expect from your experience with other programming languages, like Java, C#, and C++.

For example, you can access elements using the bracket operator [] specifying the zero-offset index for the value to retrieve.

Positivetive Indexing

In [ ]:

```
# consider an array a
a = np.array([12,23,34,56,67,78,89])
print(a)
```

In [11]:

```
# print the first element of the array a
a[0]
```

Out[11]:

12

In [15]:

```
# print 5th element of the array a
a[4]
```

Out[15]:

67

Specifying integers too large for the bound of the array will cause an error.

In [25]:

```
a[20]
```

```
---------------------------------------------------------------------------
IndexError                Traceback (most recent call last)
<ipython-input-25-2ba6acdce15e> in <module>()
----> 1 a[20]

IndexError: index 20 is out of bounds for axis 0 with size 7
```

Negative Indexing

One key difference is that you can use negative indexes to retrieve values offset from the end of the array.

For example, the index -1 refers to the last item in the array. The index -2 returns the second last item all the way back to -7 for the first item in the array a.

In [27]:

```
# print the last element of the array a
a[-1]
```

Out[27]:

89

In [30]:

```
# print the first element of the array a
```



```
a[-7]
```

Out[30]:

12

In [31]:

```
# print the second element of the array a
a[-6]
```

Out[31]:

23

Two Dimensional Indexing

Indexing two-dimensional data is similar to indexing one-dimensional data, except that a comma is used to separate the index for each dimension. **data[0,0]**

This is different from C-based languages where a separate bracket operator is used for each dimension. **data[0][0]**

For example, we can access the first row and the first column as follows:

In [35]:

```
from numpy import array
# define array
data = array([[11, 22], [33, 44], [55, 66]])
# prit array data
print(data)
# index data
print(data[0,0])
```

```
[[11 22]
 [33 44]
 [55 66]]
11
```

If we are interested in all items in the first row, we could leave the second dimension index empty, for example:

In [36]:

```
# print first row
# first method
print(data[0,])
```

```
[11 22]
```

In [43]:

```
# print first row
```



```
# second method
print(data[0,:])
```

```
[11 22]
```

In [44]:

```
# print first row
#  third  method
print(data[0,...])
```

```
[11 22]
```

In [40]:

```
# print first column
print(data[,0])  # This command does not work
```

```
  File "<ipython-input-40-d438f6cdfe29>", line 2
    print(data[,0])
               ^
SyntaxError: invalid syntax
```

In [41]:

```
# print first column
# first method
print(data[...,0])
```

```
[11 33 55]
```

In [42]:

```
# print first column
# second method
print(data[:,0])
```

```
[11 33 55]
```

In [46]:

```
# print all the rows starting from second row
print(data[1:])
```

```
[[33 44]
 [55 66]]
```

Array Slicing: Accessing Subarrays

So far, so good; creating and indexing arrays looks familiar.

Now we come to array slicing, and this is one feature that causes problems for beginners to Python and NumPy arrays.

Structures like lists and NumPy arrays can be sliced. This means that a subsequence of the structure

can be indexed and retrieved.

This is most useful in machine learning when specifying input variables and output variables, or splitting training rows from testing rows.

Slicing is specified using the colon operator ':' with a 'from' and 'to' index before and after the column respectively. The slice extends from the 'from' index and ends one item before the 'to' index.

Syntax

data[from:to]

One-Dimensional Slicing

You can access all data in an array dimension by specifying the slice ':' with no indexes.

In [48]:

```
# print all the elements of array a using ':' symbol
print(a[:])
```

```
[12 23 34 56 67 78 89]
```

The first item of the array can be sliced by specifying a slice that starts at index 0 and ends at index 4 (one item before the 'to' index).

In [49]:

```
# print the first, second, third and fourth elements (first four elements) of the array a
print(a[0:4])
```

```
[12 23 34 56]
```

We can also use negative indexes in slices. For example, we can slice the last two items in the list by starting the slice at -4 (the fourth last item) and not specifying a 'to' index; that takes the slice to the end of the dimension.

In [50]:

```
# print the last four elements of the array a
print(a[-4:])
```

```
[56 67 78 89]
```

In [51]:

```
# print the third to last elements of the array a
print(a[2:])
```

```
[34 56 67 78 89]
```

In [52]:

```
# Print all even indexed elements
print(a)
print(a[0::2])  # prints the elemts in indices 0, 2 , 4 ....
```

```
[12 23 34 56 67 78 89]
```

[12 34 67 89]

In [53]:

```
# Print all odd indexed elements
print(a)
print(a[1::2])
```

[12 23 34 56 67 78 89]

[23 56 78]

The basic slice syntax is **i:j:k** where

i is the starting index,

j is the stopping index, and

k is the step (k is not equals to 0).

In [97]:

```
# Print all elements from index 1 to index 7 with difference 2
x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
print(x)
print(x[1:7:2])
```

[0 1 2 3 4 5 6 7 8 9]

[1 3 5]

In [98]:

```
# Print all elements from index 2 to index 7 with difference 3
print(x)
print(x[2:7:3])
```

[0 1 2 3 4 5 6 7 8 9]

[2 5]

Negative i and j are interpreted as n + i and n + j where

n is the number of elements in the corresponding dimension.

In [103]:

```
# print the given array x
print(x)

# print elements of index 8 and 9
# first method
print("print elements from index 8 and 9 using positive indexing")
```



```
print(x[8:10])

# second method
print("print elements from index 8 and 9 using negitive indexing")
print(x[-2:10])
```

[0 1 2 3 4 5 6 7 8 9]

print elements from index 8 and 9 using positive indexing

[8 9]

print elements from index 8 and 9 using negitive indexing

[8 9]

Negative k makes stepping go towards smaller indices.

In [105]:

```
# print the given array x
print(x)

# print elements from index 7 to index 4 using negative k
print(x[-3:3:-1])
```

[0 1 2 3 4 5 6 7 8 9]

[7 6 5 4]

Two-Dimensional Slicing

Let's look at the two examples of two-dimensional slicing

Split Input and Output Features

It is common to split your loaded data into input variables (X) and the output variable (y).

We can do this by slicing all rows and all columns up to, but before the last column, then separately indexing the last column.

For the input features, we can select all rows and all columns except the last one by specifying ':' for in the rows index, and :-1 in the columns index.

X = [:, :-1]

For the output column, we can select all rows again using ':' and index just the last column by specifying the -1 index.

y = [:, -1]

Putting all of this together, we can separate a 3-column 2D dataset into input and output data as follows:

In [59]:


```
# split input and output
from numpy import array
# define array
data = array([[11, 22, 33], [44, 55, 66], [77, 88, 99]])
# extract first two columns
x = data[:, :-1]
# extract last column
y = data[:, -1]
print("output of given array data is:")
print(data)
print("output of print(x)")
print(x)
print("output of print(y)")
print(y)
```

output of given array data is:

[[11 22 33]

[44 55 66]

[77 88 99]]

output of print(x)

[[11 22]

[44 55]

[77 88]]

output of print(y)

[33 66 99]

Reshaping of Arrays

After slicing your data, you may need to reshape it.

For example, some libraries, such as scikit-learn, may require that a one-dimensional array of output variables (y) be shaped as a two-dimensional array with one column and outcomes for each column.

Some algorithms, like the Long Short-Term Memory recurrent neural network in Keras, require input to be specified as a three-dimensional array comprised of samples, timesteps, and features.

It is important to know how to reshape your NumPy arrays so that your data meets the expectation of specific Python libraries. We will look at these two examples.

Data Shape

NumPy arrays have a shape attribute that returns a tuple of the length of each dimension of the array.

For example:

In [61]:

```
# Running the example prints a tuple for the one dimension.
from numpy import array
# define array
data = array([11, 22, 33, 44, 55])
print(data.shape)
```

(5,)

A tuple with two lengths is returned for a two-dimensional array.

In [62]:

```
# Running the example returns a tuple with the number of rows and columns.
from numpy import array
data = array([[11, 22], [33, 44], [55, 66]])
print(data.shape)
```

(3, 2)

You can use the size of your array dimensions in the shape dimension, such as specifying parameters.

The elements of the tuple can be accessed just like an array, with the 0th index for the number of rows and the 1st index for the number of columns. For example:

In [63]:

```
# Running the example accesses the specific size of each dimension.
print('Rows: %d' % data.shape[0])
print('Cols: %d' % data.shape[1])
```

Rows: 3

Cols: 2

In [68]:

```
# print the shape of a 3-d data
data = np.zeros((2, 3, 4))
print(data)
print(data.shape)
```

(2, 3, 4)

Reshaping data using shape attribute

shape attribute can be used to **reshape** the data also

In [72]:

```
# Reshape the 3-d array y to 2-d array
data = np.zeros((2, 3, 4))
```

```
print("Given data is:")
print(data)
data.shape = (3, 8)
print("Reshaped data is")
print(data)
```

```
Given data is:
[[[0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]

 [[0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]]
Reshaped  data  is
[[0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]]
```

In [74]:

```
# Reshape the 3-d array y to 2-d array
data = np.random.random((2, 3, 4))
print("Given data is:")
print(data)
data.shape = (12, 2)
print("Reshaped data is")
print(data)
```

```
Given data is:
[[[0.98236374 0.04882206 0.65075109 0.74184213]
  [0.55915295 0.9054104  0.42296841 0.31121722]
  [0.1991875  0.07059558 0.47732611 0.0149716 ]]

 [[0.80166294 0.27110141 0.46116517 0.07375361]
  [0.95401287 0.35366554 0.7086821  0.6098683 ]
  [0.9054073  0.52688892 0.11864855 0.046618  ]]]
Reshaped data is
[[0.98236374 0.04882206]
```



```
[0.65075109 0.74184213]
 [0.55915295 0.9054104 ]
 [0.42296841 0.31121722]
 [0.1991875  0.07059558]
 [0.47732611 0.0149716 ]
 [0.80166294 0.27110141]
 [0.46116517 0.07375361]
 [0.95401287 0.35366554]
 [0.7086821  0.6098683 ]
 [0.9054073  0.52688892]
 [0.11864855 0.046618  ]]
```

In [78]:

```
# Reshape the 3-d array y to 1-d array
data = np.ones((3, 3, 2), dtype = np.int)
print("Given data is:")
print(data)
data.shape = (1, 18)
print("Reshaped data is")
print(data)
```

```
Given data is:
[[[1 1]
  [1 1]
  [1 1]]

 [[1 1]
  [1 1]
  [1 1]]

 [[1 1]
  [1 1]
  [1 1]]]
Reshaped data is
[[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]]
```

Note:

Condition for reshaping an array is the total size of new array must be unchanged. It returns an error if we try to reshape an array to a different size.

For example:

The following example returns an error because we are trying to resize the array of total size 2 *3* 5 = 30 elements to an array of total size 12 * 2 = 24 elements

In [82]:

```
# Reshape the 3-d array y to 2-d array
data = np.ones((2, 3, 5)) # total size of data is 2 * 3 * 5 = 30
print("Given data is:")
print(data)

data.shape = (12, 2) # it returns an error
print("Reshaped data is")
print(data)
```

```
Given data is:
[[[1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]]

 [[1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]]]
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-82-58a7a7b3287e> in <module>()
      4 print(data)
      5
----> 6 data.shape = (12, 2) # it returns an error
      7 print("Reshaped data is")
      8 print(data)

ValueError: cannot reshape array of size 30 into shape (12,2)
```

Reshaping data using reshape function

reshape() function gives a new shape to an array without changing its data.

For example:

In [86]:

```
a = np.arange(6).reshape((3, 2))
print(a)
```

```
[[0 1]
 [2 3]
 [4 5]]
```

In [88]:

```
# Reshape array a to another array b of size 2 by 3
b = np.reshape(a, (2, 3))
print(b)
```

```
[[0 1 2]
 [3 4 5]]
```

In [90]:

```
# Transposing the array a gives an array of size 2 by 3. However, it produces different output.
b = a.T # transposing
print(b)
```

```
[[0 2 4]
 [1 3 5]]
```

In [95]:

```
a = np.array([[1,2,3], [4,5,6]])
print("Given array is")
print(a)
# reshape the given array a to 1-d array

# C-like index ordering (row major indexing)
b = np.reshape(a, 6)
print("C-like index ordering")
print(b)

# Fortran-like index ordering (column major indexing)
print("Fortran-like index ordering")
c = np.reshape(a, 6, order='F')
print(c)
```

```
Given array is
[[1 2 3]
```

```
 [4 5 6]]
C-like index ordering
[1 2 3 4 5 6]
Fortran-like index ordering
[1 4 2 5 3 6]
```

Array Concatenation and Splitting

Concatenation

Concatenation refers to joining. This function is used to join two or more arrays of the same shape along a specified axis. The function takes the following parameters.

```
In [112]:
# concatenation of two 1-d arrays horizontally
import numpy as np
a = np.array([1,2,3,4])
b = np.array([5,6,7,8])
# cocatenate arrays a and b
conc_h = np.concatenate((a,b), axis = 0)
print(conc_h)
[1 2 3 4 5 6 7 8]
In [115]:
# concatenation of two 1-d arrays vertically
a = np.array([1,2,3,4])
b = np.array([5,6,7,8])
# cocatenate arrays a and b
conc_v = np.concatenate((a,b), axis=1)
print(conc_v)
---------------------------------------------------------------------------
AxisError                         Traceback (most recent call last)
<ipython-input-115-24f9720eb918> in <module>()
      4 b = np.array([5,6,7,8])
      5 # cocatenate arrays a and b
----> 6 conc_v = np.concatenate((a,b), axis=1)
      7 print(conc_v)

AxisError: axis 1 is out of bounds for array of dimension 1
```



```
In [123]:
# concatenation of two 2-d arrays vertically
import numpy as np
a = np.array([[1,2],[3,4]])
b = np.array([[5,6],[7,8]])
# cocatenate arrays a and b
conc_v = np.concatenate((a,b), axis=0)
print(conc_v)
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
In [124]:
# concatenation of two 2-d arrays horizontally
import numpy as np
a = np.array([[1,2],[3,4]])
b = np.array([[5,6],[7,8]])
# cocatenate arrays a and b
conc_h = np.concatenate((a,b), axis=1)
print(conc_h)
[[1 2 5 6]
 [3 4 7 8]]
In [127]:
# concatenation of two 2-d arrays having axis as none
import numpy as np
a = np.array([[1,2],[3,4]])
b = np.array([[5,6],[7,8]])
# cocatenate arrays a and b
conc_n = np.concatenate((a,b), axis=None)
print(conc_n)
[1 2 3 4 5 6 7 8]
```

Splitting

1-d arrays

```
In [137]:
# Split the array in 1 equal-sized subarrays
```



```
x = np.arange(9)
np.split(x, 1)
[array([0, 1, 2, 3, 4, 5, 6, 7, 8])]
In [134]:
# Split the array in 2 equal-sized subarrays
x = np.arange(9)
np.split(x, 2) # it throws an error because 9 elements can not be splitted into 2 equal subarrays
---------------------------------------------------------------------------
TypeError                         Traceback (most recent call last)
~\Anaconda3\lib\site-packages\numpy\lib\shape_base.py in split(ary, indices_or_sections, axis)
    777     try:
--> 778         len(indices_or_sections)
    779     except TypeError:

TypeError: object of type 'int' has no len()

During handling of the above exception, another exception occurred:

ValueError                        Traceback (most recent call last)
<ipython-input-134-33f36300c82d> in <module>()
      1 # equal splitting
      2 x = np.arange(9)
----> 3 print(np.split(x, 2))

~\Anaconda3\lib\site-packages\numpy\lib\shape_base.py in split(ary, indices_or_sections, axis)
    782         if N % sections:
    783             raise ValueError(
--> 784                 'array split does not result in an equal division')
    785     res = array_split(ary, indices_or_sections, axis)
    786     return res

ValueError: array split does not result in an equal division
In [131]:
# Split the array in 3 equal-sized subarrays
x = np.arange(9)
```



```
np.split(x, 3)
Out[131]:
[array([0, 1, 2]), array([3, 4, 5]), array([6, 7, 8])]
In [160]:
# Split the array at positions indicated in 1-D array
x = np.arange(9)
np.split(x,[4,7])
Out[160]:
[array([0, 1, 2, 3]), array([4, 5, 6]), array([7, 8])]
In [132]:
# Split the array at positions indicated in 1-D array
x = np.arange(9)
np.split(x, [2,3,4])
Out[132]:
[array([0, 1]), array([2]), array([3]), array([4, 5, 6, 7, 8])]
In [161]:
# Split the array x into three subarrays containing 2 3 and 4 elements respectively
# method-1
x = np.arange(9)
np.split(x, [2,5])
Out[161]:
[array([0, 1]), array([2, 3, 4]), array([5, 6, 7, 8])]
In [162]:
# Split the array x into three subarrays containing 2 3 and 4 elements respectively
# method-2
x = np.arange(9)
np.split(x, [2,5,9])
Out[162]:
[array([0, 1]), array([2, 3, 4]), array([5, 6, 7, 8]), array([], dtype=int32)]
```

2-d arrays

The **numpy.hsplit** is a special case of split() function where axis is 1 indicating a horizontal split regardless of the dimension of the input array.

```
In [168]:
a = np.arange(16).reshape(4,4)
```

```
print('First array:')
print(a)
print('\n')

print('Horizontal splitting:')
np.hsplit(a,2)
```

```
First array:
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]


Horizontal splitting:
Out[168]:
[array([[ 0,  1],
        [ 4,  5],
        [ 8,  9],
        [12, 13]]), array([[ 2,  3],
        [ 6,  7],
        [10, 11],
        [14, 15]])]
```

The **numpy.vsplit** is a special case of split() function where axis is 1 indicating a vertical split regardless of the dimension of the input array. The following example makes this clear.

```
In [169]:
a = np.arange(16).reshape(4,4)

print('First array:')
print(a)
print('\n')

print('Horizontal splitting:')
np.vsplit(a,2)
```

```
First array:
[[ 0  1  2  3]
```



```
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]


Horizontal splitting:
Out[169]:
[array([[0, 1, 2, 3],
        [4, 5, 6, 7]]), array([[ 8,  9, 10, 11],
        [12, 13, 14, 15]])]
In [175]:
a = np.arange(16).reshape(4,4)

print('First array:')
print(a)
print('\n')

print('Horizontal splitting:')
np.vsplit(a,3)
First array:
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]


Horizontal splitting:
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\Anaconda3\lib\site-packages\numpy\lib\shape_base.py in split(ary, indices_or_sections, axis)
    777     try:
--> 778         len(indices_or_sections)
    779     except TypeError:

TypeError: object of type 'int' has no len()
```



```
During handling of the above exception, another exception occurred:

ValueError                                Traceback (most recent call last)
<ipython-input-175-a29e75d96438> in <module>()
      6
      7 print('Horizontal splitting:')
----> 8 np.vsplit(a,3)

~\Anaconda3\lib\site-packages\numpy\lib\shape_base.py in vsplit(ary, indices_or_sections)
    897     if _nx.ndim(ary) < 2:
    898         raise ValueError('vsplit only works on arrays of 2 or more dimensions')
--> 899     return split(ary, indices_or_sections, 0)
    900
    901 def dsplit(ary, indices_or_sections):

~\Anaconda3\lib\site-packages\numpy\lib\shape_base.py in split(ary, indices_or_sections, axis)
    782         if N % sections:
    783             raise ValueError(
--> 784                 'array split does not result in an equal division')
    785     res = array_split(ary, indices_or_sections, axis)
    786     return res

ValueError: array split does not result in an equal division
In [176]:
a = np.arange(16).reshape(4,4)

print('First array:')
print(a)
print('\n')

print('Horizontal splitting:')
np.vsplit(a,4)
First array:
```



```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]


Horizontal splitting:
Out[176]:
[array([[0, 1, 2, 3]]),
 array([[4, 5, 6, 7]]),
 array([[ 8,  9, 10, 11]]),
 array([[12, 13, 14, 15]])]
In [177]:
a = np.arange(16).reshape(4,4)

print('First array:')
print(a)
print('\n')

print('Horizontal splitting:')
np.vsplit(a,[2,3])
First array:
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]


Horizontal splitting:
Out[177]:
[array([[0, 1, 2, 3],
        [4, 5, 6, 7]]), array([[ 8,  9, 10, 11]]), array([[12, 13, 14, 15]])]
```

Stacking

```
In [142]:
a = np.array((1,2,3))
```

```
b = np.array((4,5,6))
np.hstack((a,b))
```

Out[142]:

```
array([1, 2, 3, 4, 5, 6])
```

In [143]:

```
a = np.array((1,2,3))
b = np.array((4,5,6))
np.vstack((a,b))
```

Out[143]:

```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [144]:

```
a = np.array([1,2,3])
b = np.array([4,5,6])
np.vstack((a,b))
```

Out[144]:

```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [145]:

```
a = np.array([1,2,3])
b = np.array([4,5,6])
np.hstack((a,b))
```

Out[145]:

```
array([1, 2, 3, 4, 5, 6])
```

Array Aggregations

Aggregations: Min, Max, and Everything In Between

Often when faced with a large amount of data, a first step is to compute summary statistics for the data in question. Perhaps the most common summary statistics are the mean and standard deviation, which allow you to summarize the "typical" values in a dataset, but other aggregates are useful as well (the sum, product, median, minimum and maximum, quantiles, etc.).

NumPy has fast built-in aggregation functions for working on arrays; we'll discuss and demonstrate some of them here.

The following table provides a list of useful aggregation functions available in NumPy:

np.sum Compute sum of elements

np.prod Compute product of elements

np.mean Compute mean of elements

np.std Compute standard deviation

np.var Compute variance

np.min Find minimum value

np.max Find maximum value

np.argmin Find index of minimum value

np.argmax Find index of maximum value

np.median Compute median of elements

np.percentile Compute rank-based statistics of elements

np.any Evaluate whether any elements are true

np.all Evaluate whether all elements are true

1-d aggregations

In [188]:

```
a = np.array([23,87,34,90,32,334,76,12])
print("Given array is")
print(a)
print("\n")

print("sum of elements in the given array")
print(np.sum(a))    # prints sum of elements
print("\n")

print("product of elements in the given array")
print(np.prod(a))    # prints product of elements
print("\n")

print("mean of elements in the given array")
print(np.mean(a))    # prints mean of elements
print("\n")

print("standard deviation of elements in the given array")
print(np.std(a)) # prints standard deviation of elements
print("\n")

print("variance of elements in the given array")
```



```
print(np.var(a)) # prints variance of elements
print("\n")

print("minimum element in the given array")
print(np.min(a)) # prints minimum element
print("\n")
print("maximum element in the given array")
print(np.max(a))    # prints maximum element
print("\n")

print("index of mimimum element in the given array")
print(np.argmin(a))    # prints index of mimimum element
print("\n")

print("index of maximum element in the given array")
print(np.argmax(a))    # prints index of maximum element
print("\n")

print("median of elements in the given array")
print(np.median(a))    # prints median of elements
print("\n")

print("rank based statistics of elements in the given array")
print("25th percentile")
print(np.percentile(a,25))    # prints 25th percentile
print("50th percentile (median)")
print(np.percentile(a,50))    # prints 50th percentile (median)
print("75th percentile")
print(np.percentile(a,75))    # prints 75th percentile
print("\n")
```

```
Given array is
[ 23  87  34  90  32 334  76  12]


sum of elements in the given array
```



```
688


product of elements in the given array
1392390144
mean of elements in the given array
86.0


standard deviation of elements in the given array
97.90684347889069


variance of elements in the given array
9585.75


minimum element in the given array
12


maximum element in the given array
334


index of mimimum element in the given array
7


index of maximum element in the given array
5


median of elements in the given array
55.0
```

rank based statistics of elements in the given array

25th percentile

29.75

50th percentile (median)

55.0

75th percentile

87.75

In [180]:

```
# a shorter syntax is to use methods of the array object itself:
a = np.array([23,87,34,90,32,334,76,12])
print("Given array is")
print(a)
print("\n")
print("Sum of the elements in the given array is")
print(a.sum())
print("\n")
print("Minimum element in the given array is")
print(a.min())
print("\n")
print("Maximum element in the given array is")
print(a.max())
print("\n")
```

Given array is

[23 87 34 90 32 334 76 12]

Sum of the elements in the given array is

688

Minimum element in the given array is

12

Maximum element in the given array is

n [194]:

```
a = np.array([23,87,34,90,32,334,76,12])
print("Given array is")
print(a)
print("\n")

# apply some condition
b = a>50
print("logical array is")
print(b)
print("\n")
print("evaluates whether any elements are true")
print(np.any(b))    # evaluates whether any elements are true
print("\n")
print("evaluates whether all elements are true")
print(np.all(b))    # evaluates whether any elements are true
print("\n")
```

Given array is

[23 87 34 90 32 334 76 12]

logical array is

[False True False True False True True False]

evaluates whether any elements are true

True

evaluates whether all elements are true

False

multi-dimensional aggregations

In [208]:

```
x = np.array([[1, 1], [2, 2]])
print(x)
```

[[1 1]
 [2 2]]

In [210]:

```
# print column sum
x.sum(axis=0)   # columns (first dimension)
```

Out[210]:

array([3, 3])

In [211]:

```
# print sum of elements of column 1
x[:, 1].sum()
```

Out[211]:

3

In [212]:

```
# print row sum
x.sum(axis=1)   # rows (second dimension)
```

Out[212]:

array([2, 4])

In [217]:

```
# print sum of elements of row 1
x[0, :].sum()
```

Out[217]:

2

In [218]:

```
# print sum of elements of row 2
x[1, :].sum()
```

Out[218]:

4

Computations on Numpy arrays

Array arithmetic

In [195]:

```
x = np.arange(4)
print("x     =", x)
print("x + 5 =", x + 5)
print("x - 5 =", x - 5)
print("x * 2 =", x * 2)
print("x / 2 =", x / 2)
print("x // 2 =", x // 2)  # floor division
print("-x     = ", -x)
print("x ** 2 = ", x ** 2)
print("x % 2 = ", x % 2)
```

x   = [0 1 2 3]

x + 5 = [5 6 7 8]

x - 5 = [-5 -4 -3 -2]

x * 2 = [0 2 4 6]

x / 2 = [0. 0.5 1. 1.5]

x // 2 = [0 0 1 1]

-x   = [0 -1 -2 -3]

x ** 2 = [0 1 4 9]

x % 2 = [0 1 0 1]

In [226]:

```
# operations on array of different shape
a = np.array([1,1,1])
b = np.array([2,2,2])
print("a     =", a)
print("\n")
print("b     =", b)
print("\n")
print("a + b =", a + b)
print("\n")
print("a - b =", a - b)
print("\n")
print("a * b =", a * b)
print("\n")
```

```
print("a / b =", a / b)
print("\n")
print("a // b =", a // b)  # floor division
print("\n")
print("-a     = ", -a)
print("\n")
print("a ** b = ", a ** b)
print("\n")
print("a % b = ", a % b)
```

```
a     = [1 1 1]

b     = [2 2 2]

a + b = [3 3 3]

a - b = [-1 -1 -1]

a * b = [2 2 2]

a / b = [0.5 0.5 0.5]

a // b = [0 0 0]

-a     =  [-1 -1 -1]

a ** b =  [1 1 1]
a % b  =  [1 1 1]
```

In [227]:

```
# operations on array of different shape
a = np.array([1,1,1])
b = np.array([2,2,2,2])
print("a     =", a)
print("\n")
print("b     =", b)
print("\n")
print("a + b =", a + b)
print("\n")
print("a - b =", a - b)
print("\n")
```



```
print("a * b =", a * b)
print("\n")
print("a / b =", a / b)
print("\n")
print("a // b =", a // b)  # floor division
print("\n")
print("-a     = ", -a)
print("\n")
print("a ** b = ", a ** b)
print("\n")
print("a % b = ", a % b)
```

```
a     = [1 1 1]



b     = [2 2 2 2]



---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-227-e31e6a7400b6> in <module>()
      6 print("b     =", b)
      7 print("\n")
----> 8 print("a + b =", a + b)
      9 print("\n")
     10 print("a - b =", a - b)

ValueError: operands could not be broadcast together with shapes (3,) (4,)
```

Absolute value

In [196]:

```
x = np.array([-2, -1, 0, 1, 2])
print(x)
abs(x)
```

```
[-2 -1  0  1  2]
```

Out[196]:


```
array([2, 1, 0, 1, 2])
```

The corresponding NumPy ufunc is np.absolute, which is also available under the alias np.abs:

In [199]:

```
print(x)
np.abs(x)
```

```
[-2 -1  0  1  2]
```

Out[199]:

```
array([2, 1, 0, 1, 2])
```

In [200]:

```
print(x)
np.absolute(x)
```

```
[-2 -1  0  1  2]
```

Out[200]:

```
array([2, 1, 0, 1, 2])
```

In [228]:

```
a = np.array([[-1,2,-3],[4,5,6]])
np.abs(a)
```

Out[228]:

```
array([[1, 2, 3],
       [4, 5, 6]])
```

Trigonometric functions

NumPy provides a large number of useful ufuncs, and some of the most useful for the data scientist are the trigonometric functions. We'll start by defining an array of angles:

In [201]:

```
theta = np.linspace(0, np.pi, 3)
```

Now we can compute some trigonometric functions on these values:

In [203]:

```
print("theta      = ", theta)
print("sin(theta) = ", np.sin(theta))
print("cos(theta) = ", np.cos(theta))
print("tan(theta) = ", np.tan(theta))
```

```
theta      = [0.         1.57079633 3.14159265]
sin(theta) = [0.0000000e+00 1.0000000e+00 1.2246468e-16]
cos(theta) = [ 1.000000e+00 6.123234e-17 -1.000000e+00]
tan(theta) =  [ 0.00000000e+00  1.63312394e+16 -1.22464680e-16]
```


The values are computed to within machine precision, which is why values that should be zero do not always hit exactly zero. Inverse trigonometric functions are also available:

In [204]:

```
x = [-1, 0, 1]
print("x         = ", x)
print("arcsin(x) = ", np.arcsin(x))
print("arccos(x) = ", np.arccos(x))
print("arctan(x) = ", np.arctan(x))
```

```
x         =  [-1, 0, 1]
arcsin(x) =  [-1.57079633  0.          1.57079633]
arccos(x) =  [3.14159265 1.57079633 0.        ]
arctan(x) =  [-0.78539816  0.          0.78539816]
```

Exponents and logarithms

Another common type of operation available in a NumPy ufunc are the exponentials:

In [205]:

```
x = [1, 2, 3]
print("x     =", x)
print("e^x  =", np.exp(x))
print("2^x  =", np.exp2(x))
print("3^x  =", np.power(3, x))
```

```
x     = [1, 2, 3]
e^x  = [ 2.71828183  7.3890561  20.08553692]
2^x  = [2. 4. 8.]
3^x  = [ 3  9 27]
```

The inverse of the exponentials, the logarithms, are also available. The basic np.log gives the natural logarithm; if you prefer to compute the base-2 logarithm or the base-10 logarithm, these are available as well:

In [206]:

```
x = [1, 2, 4, 10]
print("x        =", x)
print("ln(x)    =", np.log(x))
print("log2(x)  =", np.log2(x))
print("log10(x) =", np.log10(x))
```

```
x        = [1, 2, 4, 10]
ln(x)    = [0.         0.69314718 1.38629436 2.30258509]
```

```
log2(x)  = [0.        1.        2.        3.32192809]
log10(x) = [0.        0.30103   0.60205999 1.        ]
```

Structured arrays

Imagine that we have several categories of data on a number of people (say, name, age, and weight), and we'd like to store these values for use in a Python program. It would be possible to store these in three separate arrays:

In [146]:

```
name = ['Alice', 'Bob', 'Cathy', 'Doug']
age = [25, 45, 37, 19]
weight = [55.0, 85.5, 68.0, 61.5]
```

In [219]:

```
# Use a compound data type for structured arrays
data = np.zeros(4, dtype={'names':('name', 'age', 'weight'), 'formats':('U10', 'i4', 'f8')})
print(data.dtype)
```

```
[('name', '<U10'), ('age', '<i4'), ('weight', '<f8')]
```

Here 'U10' translates to "Unicode string of maximum length 10," 'i4' translates to "4-byte (i.e., 32 bit) integer," and 'f8' translates to "8-byte (i.e., 64 bit) float." We'll discuss other options for these type codes in the following section.

Now that we've created an empty container array, we can fill the array with our lists of values:

In [149]:

```
data['name'] = name
data['age'] = age
data['weight'] = weight
print(data)
```

```
[('Alice', 25, 55. ) ('Bob', 45, 85.5) ('Cathy', 37, 68. )
 ('Doug', 19, 61.5)]
```

As we had hoped, the data is now arranged together in one convenient block of memory.

The handy thing with structured arrays is that you can now refer to values either by index or by name:

In [150]:

```
# Get all names
data['name']
```

Out[150]:

```
array(['Alice', 'Bob', 'Cathy', 'Doug'], dtype='<U10')
```

In [151]:

```
# Get first row of data
```



```
data[0]
```

Out[151]:

```
('Alice', 25, 55.)
```

In [152]:

```
# Get the name from the last row
data[-1]['name']
```

Out[152]:

```
'Doug'
```

In [153]:

```
# Get names where age is under 30
data[data['age'] < 30]['name']
```

Out[153]:

```
array(['Alice', 'Doug'], dtype='<U10')
```

Vectorized String Operations

One strength of Python is its relative ease in handling and manipulating string data. Pandas builds on this and provides a comprehensive set of vectorized string operations that become an essential piece of the type of munging required when working with (read: cleaning up) real-world data. In this section, we'll walk through some of the Pandas string operations, and then take a look at using them to partially clean up a very messy dataset of recipes collected from the Internet.

Introducing Pandas String Operations

We saw in previous sections how tools like NumPy and Pandas generalize arithmetic operations so that we can easily and quickly perform the same operation on many array elements. For example:

In [2]:

```
import numpy as np
x = np.array([2, 3, 5, 7, 11, 13])
x * 2
```

Out[2]:

```
array([ 4,  6, 10, 14, 22, 26])
```

This vectorization of operations simplifies the syntax of operating on arrays of data: we no longer have to worry about the size or shape of the array, but just about what operation we want done. For arrays of strings, NumPy does not provide such simple access, and thus you're stuck using a more verbose loop syntax:

In [3]:

```
data = ['peter', 'Paul', 'MARY', 'gUIDO']
[s.capitalize() for s in data]
```


Out[3]:

```
['Peter', 'Paul', 'Mary', 'Guido']
```

This is perhaps sufficient to work with some data, but it will break if there are any missing values. For example:

In [4]:

```
data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
[s.capitalize() for s in data]
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-4-3b0264c38d59> in <module>()
      1 data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
----> 2 [s.capitalize() for s in data]

<ipython-input-4-3b0264c38d59> in <listcomp>(.0)
      1 data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
----> 2 [s.capitalize() for s in data]

AttributeError: 'NoneType' object has no attribute 'capitalize'
```

Pandas includes features to address both this need for vectorized string operations and for correctly handling missing data via the str attribute of Pandas Series and Index objects containing strings. So, for example, suppose we create a Pandas Series with this data:

In [6]:

```
import pandas as pd
names = pd.Series(data)
names
```

Out[6]:

```
0    peter
1     Paul
2     None
3     MARY
4    gUIDO
dtype: object
```

We can now call a single method that will capitalize all the entries, while skipping over any missing values:

In [7]:

```
names.str.capitalize()
```

Out[7]:

```
0    Peter
```



```
1     Paul
2     None
3     Mary
4    Guido
dtype: object
```

In [8]:

```
# convert all alphabets in to lower case
names.str.lower()
```

Out[8]:

```
0    peter
1     paul
2     None
3     mary
4    guido
dtype: object
```

In [9]:

```
# convert all alphabets in to upper case
names.str.upper()
```

Out[9]:

```
0    PETER
1     PAUL
2     None
3     MARY
4    GUIDO
dtype: object
```

In [10]:

```
# Swap the case of alphabets (convert upper case alphabets to lower case and vice-versa)
names.str.swapcase()
```

Out[10]:

```
0    PETER
1     pAUL
2     None
3     mary
4    Guido
```

5 dtype:
objectIn
[11]:
```
# find the length of each string
names.str.len()
```
Out[11]:
```
0 5.0
1 4.0
2 NaN
3 4.0
4 5.0
dtype: float64
```
Tables of Pandas String Methods

If you have a good understanding of string manipulation in Python, most of Pandas string syntax is intuitive enough that it's probably sufficient to just list a table of available methods; we will start with that here, before diving deeper into a few of the subtleties. The examples in this section use the following series of names:

In [12]:
```
monte = pd.Series(['Graham Chapman', 'John Cleese', 'Terry Gilliam',
                   'Eric Idle', 'Terry Jones', 'Michael Palin'])
```
Methods similar to Python string methods

Nearly all Python's built-in string methods are mirrored by a Pandas vectorized string method. Here is a list of Pandas str methods that mirror Python string methods:

<img src = http://localhost:8888/files/str_fns1.png>

Notice that these have various return values. Some, like lower(), return a series of strings:

In [13]:
```
monte.str.lower()
```
Out[13]:
```
0 graham chapman
1 john cleese
2 terry gilliam
3 eric idle
4 terry jones
5 michael palin
```


6 dtype: object
But some others return numbers:
In [16]:
```
monte.str.len() # space is also counted as a character
```
Out[16]:
```
0 14
1 11
2 13
3 9
4 11
5 13
dtype: int64
```
In [15]:
```
monte.str.startswith('T')
```
Out[15]:
```
0 False
1 False
2 True
3 False
4 True
5 False
dtype: bool
```
In [18]:
```
monte.str.startswith('t') # Python is case-sensitive
```
Out[18]:
```
0 False
1 False
2 False
3 False
4 False
5 False
dtype: bool
```
In [20]:
```
monte.str.endswith('e')
```


Out[20]:
```
0 False
1 True
2 False
3 True
4 False
5 False
dtype: bool
```
Still others return lists or other compound values for each element:

In [21]:
```
monte.str.split() # default split condition is space
```
Out[21]:
```
0 [Graham, Chapman]
1 [John, Cleese]
2 [Terry, Gilliam]
3 [Eric, Idle]
4 [Terry, Jones]
5 [Michael, Palin]
dtype: object
```
In [22]:
```
# split the string when alphabet 'a' occurs in a string
monte.str.split('a') # 'a' will not be printed
```
Out[22]:
```
0 [Gr, h, m Ch, pm, n]
1 [John Cleese]
2 [Terry Gilli, m]
3 [Eric Idle]
4 [Terry Jones]
5 [Mich, el P, lin]
dtype: object
```
In [24]:
```
# replace 'a' with '@'
monte.str.replace('a','@')
```
Out[24]:
```
0 Gr@h@m Ch@pm@n
```


1 John Cleese
2 Terry Gilli@m
3 Eric Idle
4 Terry Jones
5 Mich@el P@lin
dtype: object
Vectorized item access and slicing

The get() and slice() operations, in particular, enable vectorized element access from each array. For example, we can get a slice of the first three characters of each array using str.slice(0, 3). Note that this behavior is also available through Python's normal indexing syntax–for example, df.str.slice(0, 3) is equivalent to df.str[0:3]:

In [25]:
```
monte.str[0:3]
```
Out[25]:
```
0 Gra
1 Joh
2 Ter
3 Eri
4 Ter
5 Mic
dtype: object
```
Indexing via df.str.get(i) and df.str[i] is likewise similar.

These get() and slice() methods also let you access elements of arrays returned by split(). For example, to extract the last name of each entry, we can combine split() and get():

In [26]:
```
monte.str.split().str.get(-1)
```
Out[26]:
```
0 Chapman
1 Cleese
2 Gilliam
3 Idle
4 Jones
5 Palin
dtype: object
```
In [27]:

```
monte.str.split().str.get(0)
```

Out[27]:

```
0     Graham
1       John
2      Terry
3       Eric
4      Terry
5    Michael
dtype: object
```

Introduction to Pandas

Topics to be covered are:

- Introduction to Pandas objects
- Data Indexing & Selection
- Operating on Data in Pandas
- Handling missing data
- Hierarchical Indexing
- Vectorized String Operations
- Visualization with Matplotlib

Python Pandas - Introduction

Pandas is an open-source Python Library providing high-performance data manipulation and analysis tool using its powerful data structures. The name Pandas is derived from the word Panel Data – an Econometrics from Multidimensional data.

In 2008, developer Wes McKinney started developing pandas when in need of high performance, flexible tool for analysis of data.

Prior to Pandas, Python was majorly used for data munging and preparation. It had very little contribution towards data analysis. Pandas solved this problem. Using Pandas, we can accomplish five typical steps in the processing and analysis of data, regardless of the origin of data — load, prepare, manipulate, model, and analyze.

Python with Pandas is used in a wide range of fields including academic and commercial domains including finance, economics, Statistics, analytics, etc.

Key Features of Pandas

- Fast and efficient DataFrame object with default and customized indexing.
- Tools for loading data into in-memory data objects from different file formats.
- Data alignment and integrated handling of missing data.
- Reshaping and pivoting of date sets

- Label-based slicing, indexing and subsetting of large data sets.
- Columns from a data structure can be deleted or inserted.
- Group by data for aggregation and transformations.
- High performance merging and joining of data.
- Time Series functionality.

Introduction to Pandas objects

At the very basic level, Pandas objects can be thought of as enhanced versions of NumPy structured arrays in which the rows and columns are identified with labels rather than simple integer indices.

The three fundamental Pandas data structures:

- Series
- DataFrame
- Index.

We will start our code sessions with the standard NumPy and Pandas imports:

In [2]:

```
import numpy as np
import pandas as pd
```

The Pandas Series Object

A Pandas Series is a one-dimensional array of indexed data.

pd.Series(data, index=index)

where index is an optional argument, and data can be one of many entities.

For example, data can be a list or NumPy array, in which case index defaults to an integer sequence:

In [10]:

```
# Create a numpy array
np_array = np.array([0.25, 0.5, 0.75, 1.0])

# Create a pandas series object
data =  pd.Series(np_array)
data
```

Out[10]:

```
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64
```

Attributes of Series object

index

Index values must be unique and hashable, same length as data. Default np.arrange(n) if no index is passed.

dtype

dtype is for data type. If None, data type will be inferred

In [43]:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0])
print(data)
print("\n")
# Print the attributes of Series object data
print(data.index)
print("\n")
print(data.dtype)
```

```
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64



RangeIndex(start=0, stop=4, step=1)



float64
```

Explicit indexing

From what we've seen so far, it may look like the Series object is basically interchangeable with a one-dimensional NumPy array. The essential difference is the presence of the index: while the Numpy Array has an implicitly defined integer index used to access the values, the Pandas Series has an explicitly defined index associated with the values.

This explicit index definition gives the Series object additional capabilities. For example, the index need not be an integer, but can consist of values of any desired type. For example, if we wish, we can use strings as an index:

In [6]:

```
# implicit indexing
data = pd.Series([0.25, 0.5, 0.75, 1.0])
```



```
print("implicit indexing")
print(data)
print("\n")

# explicit indexing
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
print("explicit indexing")
data
```

```
implicit indexing
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64



explicit indexing
```

Out[6]:

```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

We can even use non-contiguous or non-sequential indices:

In [7]:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=[2, 5, 3, 7])
data
```

Out[7]:

```
2    0.25
5    0.50
3    0.75
7    1.00
dtype: float64
```

Series as specialized dictionary

In this way, you can think of a Pandas Series a bit like a specialization of a Python dictionary. A dictionary is a structure that maps arbitrary keys to a set of arbitrary values, and a Series is a structure which maps typed keys to a set of typed values. This typing is important: just as the type-specific compiled code behind a NumPy array makes it more efficient than a Python list for certain operations, the type information of a Pandas Series makes it much more efficient than Python dictionaries for certain operations.

The Series-as-dictionary analogy can be made even more clear by constructing a Series object directly from a Python dictionary:

In [8]:

```
population_dict = {'California': 38332521,
                   'Texas': 26448193,
                   'New York': 19651127,
                   'Florida': 19552860,
                   'Illinois': 12882135}
population = pd.Series(population_dict)
population
```

Out[8]:

```
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
dtype: int64
```

The Pandas Series Object

The next fundamental structure in Pandas is the DataFrame. Like the Series object discussed in the previous section, the DataFrame can be thought of either as a generalization of a NumPy array, or as a specialization of a Python dictionary. We'll now take a look at each of these perspectives.

In [24]:

```
# create a pandas series object population_dist
population_dict = {'California': 38332521,
                   'Texas': 26448193,
                   'New York': 19651127,
                   'Florida': 19552860,
                   'Illinois': 12882135}
```



```
population =
pd.Series(population_dict)population
```

Out[24]:

```
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
dtype: int64
```

In [28]:

```
# create a pandas data frame simple_df using the above pandas series object population_dist
simple_df = pd.DataFrame(population)
simple_df
```

Out[28]:

	0
California	38332521
Texas	26448193
New York	19651127
Florida	19552860
Illinois	12882135

In [29]:

```
# create a pandas data frame simple_df using the above pandas series object population_dist
# name the column as 'Polpulation'
simple_df = pd.DataFrame(population,columns=['Population'])
simple_df
```

Out[29]:

	Population
California	38332521

Texas	26448193
New York	19651127
Florida	19552860
Illinois	12882135

In [31]:

```
# create a pandas series object population_dist
population_dict = {'California': 38332521,
                   'Texas': 26448193,
                   'New York': 19651127,
                   'Florida': 19552860,
                   'Illinois': 12882135}
population = pd.Series(population_dict)

# print the pandas series object population_dist
population
```

Out[31]:

```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
dtype: int64
```

In [33]:

```
# create a pandas series object area_dist
area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
             'Florida': 170312, 'Illinois': 149995}
area = pd.Series(area_dict)

# create a pandas series object area_dist
area
```

Out[33]:

```
California    423967
```



```
Texas         695662
New York      141297
Florida       170312
Illinois      149995
dtype: int64
```

Now that we have this along with the population Series from before, we can use a dictionary to construct a single two-dimensional object containing this information:

In [38]:

```
states = pd.DataFrame([population, area],columns=['population','area'])
states
```

Out[38]:

	population	area
0	NaN	NaN
1	NaN	NaN

In [25]:

```
states = pd.DataFrame({'population': population,
                       'area': area})
states
```

Out[25]:

	population	area
California	38332521	423967
Texas	26448193	695662
New York	19651127	141297
Florida	19552860	170312
Illinois	12882135	149995

Given a two-dimensional array of data, we can create a DataFrame with any specified column and index names. If omitted, an integer index will be used for each:

In [46]:

```
# From a two-dimensional NumPy array
```

```
# create a 2-d numpy array
data = np.random.rand(3, 2)

# create a pandas data frame my_df from 2-d numpy array data

my_df = pd.DataFrame(data,
          columns=['foo', 'bar'],
          index=['a', 'b', 'c'])

# print the data frame my_df
my_df
```

Out[46]:

	foo	bar
a	0.888558	0.793189
b	0.368391	0.464119
c	0.390875	0.344930

Attributes of DataFrame object

index

For the row labels, the Index to be used for the resulting frame is Optional Default np.arrange(n) if no index is passed.

columns

For column labels, the optional default syntax is - np.arrange(n). This is only true if no index is passed.

dtype

Data type of each column.

In [42]:

```
states = pd.DataFrame({'population': population,
                       'area': area})
states
print(states)
print("\n")
# Print the attributes of Series object data
```



```
print(states.index)
print("\n")
print(states.columns)
print("\n")
print(states.population.dtype)
print("\n")
print(states.area.dtype)
```

```
            population    area
California    38332521  423967
Texas         26448193  695662
New York      19651127  141297
Florida       19552860  170312
Illinois      12882135  149995


Index(['California', 'Texas', 'New York', 'Florida', 'Illinois'], dtype='object')


Index(['population', 'area'], dtype='object')

int64

int64
```

The Pandas Index Object

We have seen here that both the Series and DataFrame objects contain an explicit index that lets you reference and modify data.

This Index object is an interesting structure in itself, and it can be thought of either as an immutable array or as an ordered set (technically a multi-set, as Index objects may contain repeated values). Those views have some interesting consequences in the operations available on Index objects. As a simple example, let's construct an Index from a list of integers:

In [47]:

```
ind = pd.Index([2, 3, 5, 7, 11])
ind
```

Out[47]:

```
Int64Index([2, 3, 5, 7, 11], dtype='int64')
```

Attributes of Pandas Index Object

In [48]:


```
# create a pandas index object
ind = pd.Index([2, 3, 5, 7, 11])
ind

# print the index object
print("size of given index is")

print(ind.size)
print("\n")
print("shape of given index is")
print(ind.shape)
print("\n")
print("No of dimensions of given index is")
print(ind.ndim)
print("\n")
print("datatype of given index is")
print(ind.dtype)
```

```
size of given index is
5
shape of given index
is(5,)

No of dimensions of given index is
1

datatype of given index is
int64
```

In [49]:

```
# index objects are immutable
ind[1] = 0
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-49-abacefeb0579> in <module>()
      1 # index objects are immutable
----> 2 ind[1] = 0
```



```
~\Anaconda3\lib\site-packages\pandas\core\indexes\base.py in __setitem__(self, key, value)
   2063
   2064     def __setitem__(self, key, value):
-> 2065         raise TypeError("Index does not support mutable operations")
   2066
   2067     def __getitem__(self, key):

TypeError: Index does not support mutable operations
```

Pandas objects are designed to facilitate operations such as joins across datasets, which depend on many aspects of set arithmetic. The Index object follows many of the conventions used by Python's built-in set data structure, so that unions, intersections, differences, and other combinations can be computed in a familiar way:

In [50]:

```
indA = pd.Index([1, 3, 5, 7, 9])
indB = pd.Index([2, 3, 5, 7, 11])
```

In [51]:

```
indA & indB  # intersection
```

Out[51]:

```
Int64Index([3, 5, 7], dtype='int64')
```

In [52]:

```
indA | indB  # union
```

Out[52]:

```
Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')
```

Data Indexing & Selection

In the previous module, we looked in detail at methods and tools to access, set, and modify values in NumPy arrays. These included

indexing (e.g., arr[2, 1]),

slicing (e.g., arr[:, 1:5]),

masking (e.g., arr[arr > 0]),

fancy indexing (e.g., arr[0, [1, 5]]),

and combinations thereof (e.g., arr[:, [1, 5]]).

Here we'll look at similar means of accessing and modifying values in Pandas Series and DataFrame objects. If you have used the NumPy patterns, the corresponding patterns in Pandas will feel very

familiar, though there are a few quirks to be aware of.

We'll start with the simple case of the one-dimensional Series object, and then move on to the more complicated two-dimesnional DataFrame object.

Data Selection in Series

As we saw in the previous section, a Series object acts in many ways like a one-dimensional NumPy array, and in many ways like a standard Python dictionary. If we keep these two overlapping analogies in mind, it will help us to understand the patterns of data indexing and selection in these arrays.

Series as dictionary

Like a dictionary, the Series object provides a mapping from a collection of keys to a collection of values:

In [53]:

```
import pandas as pd
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
data
```

Out[53]:

```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

In [54]:

```
data['b']
```

Out[54]:

```
0.5
```

In [56]:

```
data[0]
```

Out[56]:

```
0.25
```

Series objects can even be modified with a dictionary-like syntax. Just as you can extend a dictionary by assigning to a new key, you can extend a Series by assigning to a new index value:

In [58]:

```
# adding new value to a Series object
data['e'] = 1.25
```



```
data
```

Out[58]:

```
a    0.25
b    0.50
c    0.75
d    1.00
e    1.25
dtype: float64
```

Series as one-dimensional array

A Series builds on this dictionary-like interface and provides array-style item selection via the same basic mechanisms as NumPy arrays – that is, slices, masking, and fancy indexing. Examples of these are as follows:

In [62]:

```
# print data
print(data)
print("\n")

# slicing by explicit index
print(data['a':'c'])
```

```
a    0.25
b    0.50
c    0.75
d    1.00
e    1.25
dtype: float64



a    0.25
b    0.50
c    0.75
dtype: float64
```

In [63]:

```
# masking
print(data[(data > 0.3) & (data < 0.8)])
```

```
b    0.50
```



```
c    0.75
dtype: float64
```

In [64]:

```
# fancy indexing
data[['a', 'e']]
```

Out[64]:

```
a    0.25
e    1.25
dtype: float64
```

Among these, slicing may be the source of the most confusion. Notice that when slicing with an explicit index (i.e., data['a':'c']), the final index is included in the slice, while when slicing with an implicit index (i.e., data[0:2]), the final index is excluded from the slice.

Indexers: loc, iloc, and ix

These slicing and indexing conventions can be a source of confusion. For example, if your Series has an explicit integer index, an indexing operation such as data[1] will use the explicit indices, while a slicing operation like data[1:3] will use the implicit Python-style index.

In [65]:

```
data = pd.Series(['a', 'b', 'c'], index=[1, 2, 3])
data
```

Out[65]:

```
1    a
2    b
3    c
dtype: object
```

In [66]:

```
# explicit index when indexing
data[1]
```

Out[66]:

```
'a'
```

In [67]:

```
# explicit index when indexing
data[2]
```

Out[67]:

```
'b'
```

In [71]:


```
# implicit index when slicing
data[1:3]
```

Out[71]:

```
2    b
3    c
dtype: object
```

Because of this potential confusion in the case of integer indexes, Pandas provides some special indexer attributes that explicitly expose certain indexing schemes. These are not functional methods, but attributes that expose a particular slicing interface to the data in the Series.

First, the **loc** attribute allows indexing and slicing that always references the **explicit index**:

In [72]:

```
# print data
data
```

Out[72]:

```
1    a
2    b
3    c
dtype: object
```

In [73]:

```
# loc references explicit index in indexing
data.loc[1]
```

Out[73]:

```
'a'
```

In [74]:

```
# loc references explicit index in indexing
data.loc[2]
```

Out[74]:

```
'b'
```

In [75]:

```
# loc references explicit index in slicing also
data.loc[1:3]
```

Out[75]:

```
1    a
2    b
3    c
```

```
dtype: object
```

The **iloc** attribute allows indexing and slicing that always references the **implicit Python-style index**:

In [78]:

```
# Print data
data
```

Out[78]:

```
1    a
2    b
3    c
dtype: object
```

In [76]:

```
# iloc references implicit index in indexing
data.iloc[1]
```

Out[76]:

```
'b'
```

In [77]:

```
# iloc references implicit index in indexing
data.iloc[2]
```

Out[77]:

```
'c'
```

In [79]:

```
# iloc references implicit index in slicing
data.iloc[1:3]
```

Out[79]:

```
2    b
3    c
dtype: object
```

Hierarchical Indexing

Up to this point we've been focused primarily on one-dimensional and two-dimensional data, stored in Pandas Series and DataFrame objects, respectively.

Often it is useful to go beyond this and store higher-dimensional data–that is, data indexed by more than one or two keys.

In this section, we'll explore the direct creation of MultiIndex objects, considerations when indexing, slicing, and computing statistics across multiply indexed data, and useful routines for converting between simple and hierarchically indexed representations of your data.

We begin with the standard imports:

In [1]:

```
import pandas as pd
import numpy as np
```

In [2]:

```
# Creation of a Series Object pop2000 representing the population in year 2000
index = ['California','New York','Texas']
populations = [33871648, 18976457, 20851820]
pop2000 = pd.Series(populations, index=index)
pop2000
```

Out[2]:

```
California    33871648
New York      18976457
Texas         20851820
dtype: int64
```

In [3]:

```
# Creation of a Series Object pop2010 representing the population in year 2010
index = ['California','New York','Texas']
populations = [37253956, 19378102, 25145561]
pop2010 = pd.Series(populations, index=index)
pop2010
```

Out[3]:

```
California    37253956
New York      19378102
Texas         25145561
dtype: int64
```

A Multiply Indexed Series

Let's start by considering how we might represent two-dimensional data within a one-dimensional Series. For concreteness, we will consider a series of data where each point has a character and numerical key.

The bad way

Suppose you would like to track data about states from two different years. Using the Pandas tools we've already covered, you might be tempted to simply use Python tuples as keys:

In [4]:

```
index = [('California', 2000), ('California', 2010),
```



```
         ('New York', 2000), ('New York', 2010),
         ('Texas', 2000), ('Texas', 2010)]
populations = [33871648, 37253956,
               18976457, 19378102,
               20851820, 25145561]
pop = pd.Series(populations, index=index)
pop
```

Out[4]:

```
(California, 2000)    33871648
(California, 2010)    37253956
(New York, 2000)      18976457
(New York, 2010)      19378102
(Texas, 2000)         20851820
(Texas, 2010)         25145561
dtype: int64
```

With this indexing scheme, you can straightforwardly index or slice the series based on this multiple index:

In [5]:

```
pop[('California', 2010):('Texas', 2000)]
```

Out[5]:

```
(California, 2010)    37253956
(New York, 2000)      18976457
(New York, 2010)      19378102
(Texas, 2000)         20851820
dtype: int64
```

But the convenience ends there. For example, if you need to select all values from 2010, you'll need to do some messy (and potentially slow) munging to make it happen:

In [6]:

```
pop[[i for i in pop.index if i[1] == 2010]]
```

Out[6]:

```
(California, 2010)    37253956
(New York, 2010)      19378102
(Texas, 2010)         25145561
dtype: int64
```

This produces the desired result, but is not as clean (or as efficient for large datasets) as the slicing

syntax we've grown to love in Pandas.

The Better Way: Pandas MultiIndex

Fortunately, Pandas provides a better way. Our tuple-based indexing is essentially a rudimentary multi-index, and the Pandas MultiIndex type gives us the type of operations we wish to have. We can create a multi-index from the tuples as follows:

In [7]:

```
index = pd.MultiIndex.from_tuples(index)
index
```

Out[7]:

```
MultiIndex(levels=[['California', 'New York', 'Texas'], [2000, 2010]],
           labels=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]])
```

Notice that the MultiIndex contains multiple levels of indexing–in this case, the state names and the years, as well as multiple labels for each data point which encode these levels.

If we re-index our series with this MultiIndex, we see the hierarchical representation of the data:

In [8]:

```
pop = pop.reindex(index)
pop
```

Out[8]:

```
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
```

Here the first two columns of the Series representation show the multiple index values, while the third column shows the data. Notice that some entries are missing in the first column: in this multi-index representation, any blank entry indicates the same value as the line above it.

Now to access all data for which the second index is 2010, we can simply use the Pandas slicing notation:

In [10]:

```
pop[:, 2010]
```

Out[10]:

```
California    37253956
New York      19378102
```

```
Texas       25145561
dtype: int64
```

In [12]:

```
pop['California']
```

Out[12]:

```
2000    33871648
2010    37253956
dtype: int64
```

The result is a singly indexed array with just the keys we're interested in. This syntax is much more convenient (and the operation is much more efficient!) than the home-spun tuple-based multi-indexing solution that we started with. We'll now further discuss this sort of indexing operation on hiearchically indexed data.

convert a multiply indexed Series into a conventionally indexed DataFrame (MultiIndex as extra dimension)

You might notice something else here: we could easily have stored the same data using a simple DataFrame with index and column labels. In fact, Pandas is built with this equivalence in mind. The unstack() method will quickly convert a multiply indexed Series into a conventionally indexed DataFrame:

In [15]:

```
# print pop
print("Mulply indexed Series Object 'pop'")
print("\n")
print(pop)
print("\n")

# convert series object 'pop' to data frame object
pop_df = pop.unstack()

# print pop_df
print("conventionally indexed DataFrame Object 'pop_df'")
print("\n")
print(pop_df)
```

```
Mulply indexed Series Object 'pop'


California  2000    33871648
```



```
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64


conventionally indexed DataFrame Object 'pop_df'


              2000      2010
California  33871648  37253956
New York    18976457  19378102
Texas       20851820  25145561
```

Naturally, the stack() method provides the opposite operation:

In [16]:

```
pop_df.stack()
```

Out[16]:

```
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
```

Methods of MultiIndex Creation

The most straightforward way to construct a multiply indexed Series or DataFrame is to simply pass a list of two or more index arrays to the constructor. For example:

In [17]:

```
df = pd.DataFrame(np.random.rand(4, 2),
                  index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
                  columns=['data1', 'data2'])
df
```

Out[17]:

		data1	data2
a	**1**	0.968206	0.439655
	2	0.856870	0.239278
b	**1**	0.706952	0.351686
	2	0.951020	0.157524

The work of creating the MultiIndex is done in the background.

Similarly, if you pass a dictionary with appropriate tuples as keys, Pandas will automatically recognize this and use a MultiIndex by default:

In [18]:

```
data = {('California', 2000): 33871648,
        ('California', 2010): 37253956,
        ('Texas', 2000): 20851820,
        ('Texas', 2010): 25145561,
        ('New York', 2000): 18976457,
        ('New York', 2010): 19378102}
pd.Series(data)
```

Out[18]:

```
California  2000    33871648
            2010    37253956
Texas       2000    20851820
            2010    25145561
New York    2000    18976457
            2010    19378102
dtype: int64
```

MultiIndex level names

Sometimes it is convenient to name the levels of the MultiIndex. This can be accomplished by passing the names argument to any of the above MultiIndex constructors, or by setting the names attribute of the index after the fact:

In [19]:

```
pop.index.names = ['state', 'year']
pop
```

Out[19]:


```
state       year
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
```

Indexing and Slicing a MultiIndex

Indexing and slicing on a MultiIndex is designed to be intuitive, and it helps if you think about the indices as added dimensions. We'll first look at indexing multiply indexed Series, and then multiply-indexed DataFrames.

In [20]:

```
pop
```

Out[20]:

```
state       year
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
```

We can access single elements by indexing with multiple terms:

In [21]:

```
pop['California', 2000]
```

Out[21]:

```
33871648
```

The MultiIndex also supports partial indexing, or indexing just one of the levels in the index. The result is another Series, with the lower-level indices maintained:

In [22]:

```
pop['California']
```

Out[22]:

```
year
```

```
2000    33871648
2010    37253956
dtype: int64
```

Partial slicing is available as well, as long as the MultiIndex is sorted

In [23]:

```
pop.loc['California':'New York']
```

Out[23]:

```
state       year
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
dtype: int64
```

With sorted indices, partial indexing can be performed on lower levels by passing an empty slice in the first index:

In [24]:

```
pop[:, 2000]
```

Out[24]:

```
state
California    33871648
New York      18976457
Texas         20851820
dtype: int64
```

Other types of indexing and selection (discussed in Data Indexing and Selection) work as well; for example, selection based on Boolean masks:

In [25]:

```
pop[pop > 22000000]
```

Out[25]:

```
state       year
California  2000    33871648
            2010    37253956
Texas       2010    25145561
dtype: int64
```

Selection based on fancy indexing also works:

In [26]:

```
pop[['California', 'Texas']]
```

Out[26]:

```
state       year
California  2000    33871648
            2010    37253956
Texas       2000    20851820
            2010    25145561
dtype: int64
```

Visualization with Matplotlib

Matplotlib is a multi-platform data visualization library built on NumPy arrays, and designed to work with the broader SciPy stack.

One of Matplotlib's most important features is its ability to play well with many operating systems and graphics backends.

Matplotlib supports dozens of backends and output types, which means you can count on it to work regardless of which operating system you are using or which output format you wish.

This cross-platform, everything-to-everyone approach has been one of the great strengths of Matplotlib. It has led to a large user base, which in turn has led to an active developer base and Matplotlib's powerful tools and ubiquity within the scientific Python world.

Importing Matplotlib

Just as we use the np shorthand for NumPy and the pd shorthand for Pandas, we will use some standard shorthands for Matplotlib imports:

In [1]:

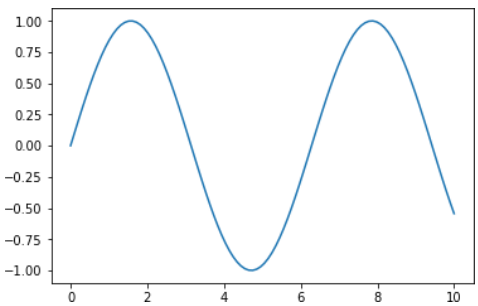
```
import matplotlib as mpl

import matplotlib.pyplot as plt
```

The plt interface is what we will use most often

Drawing a simple Line plot

In [4]:


```
import matplotlib.pyplot as plt

import numpy as np


x = np.linspace(0, 10, 100)
# Drawing sine curve

plt.plot(x, np.sin(x))


plt.show()
```



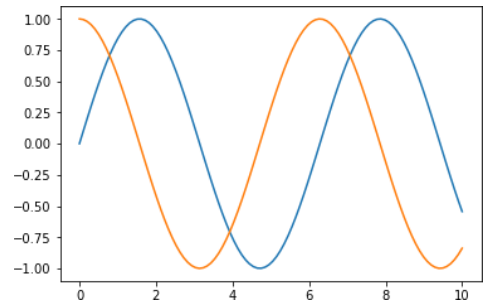
Drawing Multi-line Line plot

In [5]:

```
x = np.linspace(0, 10, 100)
```



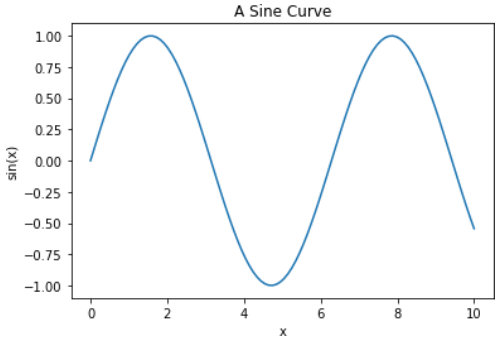
```
# Drawing both sine and cosine curves on same plot

plt.plot(x, np.sin(x))

plt.plot(x, np.cos(x))


plt.show()
```



Adding Labels

```
x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x))

# Adding title of graph

plt.title("A Sine Curve")

# Adding label of x-axis and y-axis

plt.xlabel("x")

plt.ylabel("sin(x)")
```

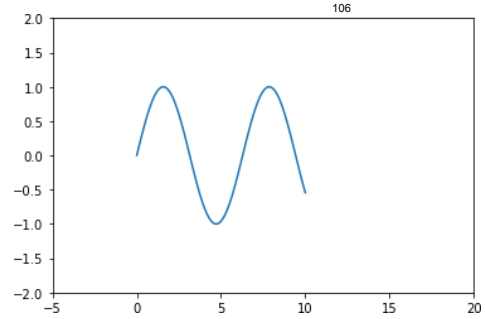
```
plt.show()
```



Adding Limits

In [22]:

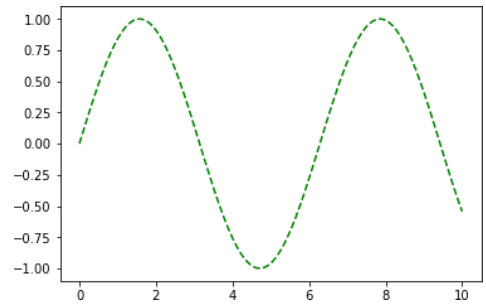
```
x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x))


# Adding limits of x-axis and y-axis

plt.xlim(-5, 20)

plt.ylim(-2, 2);


plt.show()
```

Specifying Line Color and Line Type

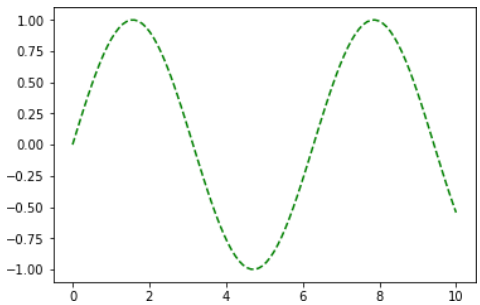
In [9]:

```
x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x), color = 'green', linestyle='dashed')

plt.show()
```

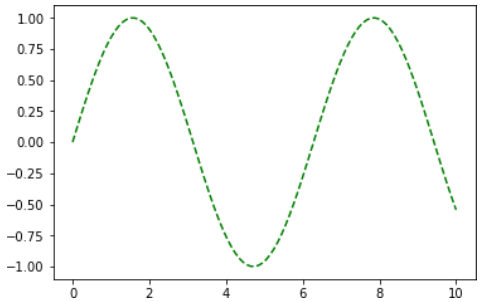


In [17]:


```
x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x), color = 'g', linestyle='--')

plt.show()
```

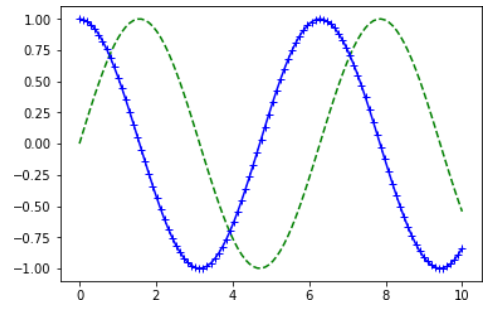


```
x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x), 'g--')

plt.show()
```



In [21]:

```
x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x), 'g--')

plt.plot(x, np.cos(x), 'b-+')

plt.show()
```

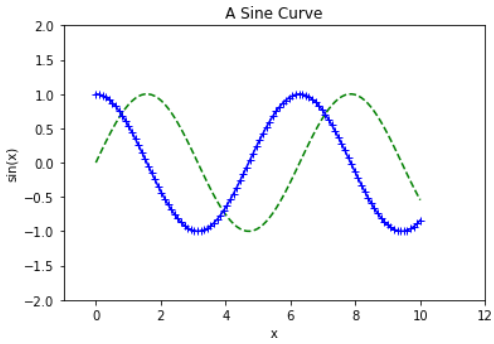


Drawing a graph with all attributes mentioned above

In [30]:

```
x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x), 'g--')

plt.plot(x, np.cos(x), 'b-+')

# Adding limits of x-axis and y-axis

plt.xlim(-1, 12)

plt.ylim(-2, 2);
```

```
# Adding title of graph

plt.title("A Sine Curve")


# Adding label of x-axis and y-axis

plt.xlabel("x")

plt.ylabel("sin(x)")

plt.show()
```



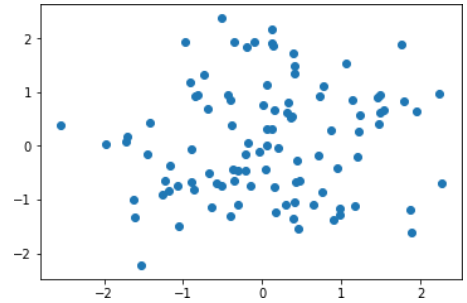
Drawing a Scatter Plot

In [26]:

```
rng = np.random.RandomState(0)

x = rng.randn(100)

y = rng.randn(100)
```

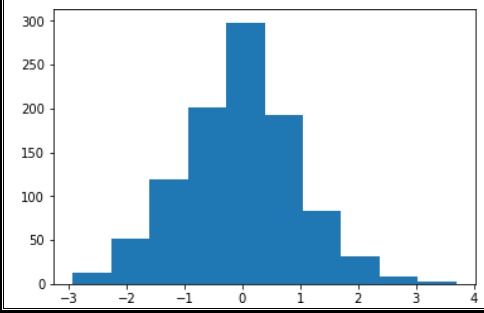


```
plt.scatter(x, y)

plt.show()
```



Drawing a Histogram

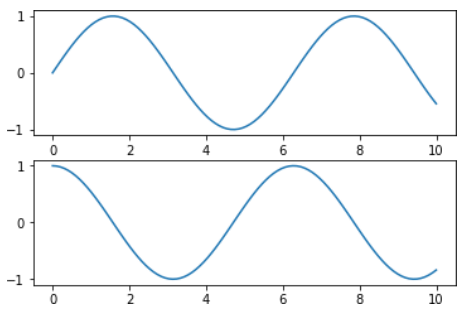
```
data = np.random.randn(1000)

plt.hist(data)

plt.show()
```



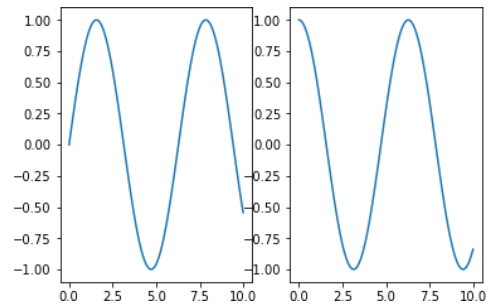
sub-graphs

In [32]:

```
# Draw a 2 by 1 graph

x = np.linspace(0, 10, 100)

# create the first of two panels and set current axis

plt.subplot(2, 1, 1) # (rows, columns, panel number)

plt.plot(x, np.sin(x))

# create the second panel and set current axis

plt.subplot(2, 1, 2)

plt.plot(x, np.cos(x));
```




```
# Draw a 1 by 2 graph

x = np.linspace(0, 10, 100)


# create the first of two panels and set current axis

plt.subplot(1, 2, 1) # (rows, columns, panel number)

plt.plot(x, np.sin(x))

# create the second panel and set current axis

plt.subplot(1, 2, 2)

plt.plot(x, np.cos(x));
```



```
# Draw a 2 by 2 graph

x = np.linspace(0, 10, 100)

# create the first panel and set current axis

plt.subplot(2, 2, 1) # (rows, columns, panel number)

plt.plot(x, np.sin(x))

# create the second panel and set current axis

plt.subplot(2, 2, 2)

plt.plot(x, np.cos(x));

# create the third panel and set current axis

plt.subplot(2, 2, 3)

rng = np.random.RandomState(0)

x = rng.randn(100)

y = rng.randn(100)

plt.scatter(x, y)

# create the fourth panel and set current axis

plt.subplot(2, 2, 4)

data = np.random.randn(1000)

plt.hist(data)
```

Out[34]:

```
(array([ 10.,  36.,  91., 178., 220., 213., 148.,  61.,  38.,   5.]),
 array([-2.79172089, -2.21836312, -1.64500535, -1.07164759, -0.49828982,
         0.07506795,  0.64842571,  1.22178348,  1.79514125,  2.36849901,
```



```
         2.94185678]),

<a list of 10 Patch objects>)
```